HOCHSCHULE FÜR TECHNIK STUTTGART

Rocket App

Pervasive Computing

Autoren:

Paul Durz	22dupa1bif@hft-stuttgart.de
	Matrikelnummer: 1005080
Dominik Gräter	22grdo1bif@hft-stuttgart.de
	Matrikelnummer: 1005093
Manuel Holm	22homa1bif@hft-stuttgart.de
	Matrikelnummer: 1005094
Ron Seifried	22sero1bif@hft-stuttgart.de
	Matrikelnummer: 1005466

Hochschule für Technik Stuttgart
Schellingstraße 24, 70174 Stuttgart, Deutschland
info@hft-stuttgart.de
https://www.hft-stuttgart.de/

Abgabedatum: June 26, 2025

Contents

List of Figures

1 Einleitung

Im Rahmen des Moduls Ubiquitous Computing von Professor Knauth sollten die Studierenden das Erlernte in einem praktischen Projekt anwenden. Das Projekt beschäftigt sich mit der Thematik des Ubiquitous Computings. Die Durchdringung von Alltagsgegenständen mit Informationstechnologie und die Vernetzung von smarten Systemen führen zu einer Umwelt, in der in allen Bereichen Veränderungen auftreten. Wir haben uns dabei für eine Mobile App für Smartphones entschieden.

1.1 Projektbeschreibung

Die **Rocket App** ist eine mobile Anwendung, die es Nutzerinnen und Nutzern ermöglicht, ihre täglichen Schritte automatisch zu erfassen. Die App belohnt Aktivität durch ein Punktesystem, das sowohl auf der Anzahl der Schritte als auch auf der Teilnahme an Herausforderungen (Challenges) basiert. Zusätzlich können Läufe (Runs) aufgezeichnet, auf Karten dargestellt und im Nachhinein analysiert werden. Die Kombination aus Bewegungstracking, Gamification und sozialem Vergleich macht die Rocket App zu einem motivierenden Begleiter im Alltag.

1.2 Ziele und Motivation

Ziel der Rocket App ist es, Nutzerinnen und Nutzer zu einem aktiveren Lebensstil zu motivieren. Durch spielerische Elemente wie Punkte, Ranglisten und Challenges soll Bewegung im Alltag gefördert und langfristig zur Gewohnheit gemacht werden. Gerade im Kontext zunehmender Digitalisierung und Bewegungsmangel bietet die App eine niedrigschwellige Möglichkeit, körperliche Aktivität zu fördern und messbar zu machen.

1.3 Website

Für die Möglichkeit, die App mit einer Website erweitern zu dürfen, danken wir Professor Dr. Knauth und Prof. Dr. Mosler. Diese Erweiterungen und die Darstellung der App auf einer Website sind nicht Teil der Pervasive Computing Vorlesung, gehören aber zu unserem Projekt dazu und sollten hier nicht unerwähnt bleiben.

Die Website bietet dieselben Features wie die App, man kann aber auch noch Runs planen, man hat einen globalen Chat und man hat die Möglichkeit, die App zu downloaden. Die Website ist über diesen Link einsehbar `https://rocket-app.social`.

Die Zielgruppe der App sind Menschen, die ihre Bewegung im Alltag tracken und sich selbst oder im Wettbewerb mit anderen motivieren möchten. Dazu gehören sowohl Gelegenheitsnutzer als auch sportlich Aktive.

2 Entwicklung

2.1 Versionsverwaltung

Für die Entwicklung der **Rocket App** wurde ein zentrales Git-Repository unter GitHub verwendet. Im Zuge dessen haben wir eine eigene Organisation namens `RealTeamRocket` auf GitHub gegründet, um die Projektressourcen zentral zu verwalten und die Zusammenarbeit im Team zu erleichtern. Das Repository ist unter folgendem Link öffentlich einsehbar: `https://github.com/RealTeamRocket/rocket-app`

Das Repository enthält sämtliche Quellcodes des Projekts – darunter das Flutter-Frontend der App, das Backend sowie eine begleitende Website. Durch diese zentrale Struktur wird sichergestellt, dass alle Komponenten konsistent versioniert und gepflegt werden können.

Zur effizienten Zusammenarbeit im Team wurde mit einem **Branching-Modell** gearbeitet. Dabei existieren Hauptzweige wie `main` und `dev`, während neue Features oder Bugfixes in separaten Feature-Branches entwickelt und anschließend via Pull Request integriert wurden. Dieses Vorgehen erlaubt paralleles Arbeiten, minimiert Merge-Konflikte und sorgt für eine saubere Trennung zwischen stabilen Releases und laufender Entwicklung.

Zur Sprint-Planung und Aufgabenverteilung kommt zusätzlich ein **GitHub Project Board** zum Einsatz, das als Kanban-Board konfiguriert wurde. Dort werden Tickets angelegt, priorisiert und den jeweiligen Teammitgliedern zugewiesen. Das Board dient als zentrale Planungsgrundlage für unsere zweiwöchigen Sprints und bietet eine transparente Übersicht über den Fortschritt einzelner Aufgaben und Meilensteine.

Durch die Kombination aus GitHub, strukturiertem Branch-Workflow und projektbezogener Aufgabenplanung konnte eine effektive und nachvollziehbare Versionskontrolle realisiert werden, die sowohl technische als auch organisatorische Anforderungen erfüllt.

2.2 Entwicklungsumgebungen

Für die Entwicklung der **Rocket App** kamen unterschiedliche Entwicklungsumgebungen zum Einsatz, abgestimmt auf die jeweiligen Anforderungen der Frontend- und Backend-Komponenten.

Die App wurde mit dem Framework **Flutter**[4] entwickelt. Für die mobile Entwicklung wurde hauptsächlich **Android Studio**[1] verwendet, insbesondere für das Erstellen, Debuggen und Testen auf realen Geräten sowie Emulatoren. Ergänzend kam auch der integrierte **Emulator von Visual Studio Code (VS Code)**[23] zum Einsatz, insbesondere für schnelle Tests und kleinere Anpassungen während der Entwicklung. Dank der Flutter-Integration in beide Umgebungen konnten Entwickler flexibel je nach Präferenz arbeiten.

Für das **Backend**, das in der Programmiersprache **Go**[11] entwickelt wurde, war die Wahl der Entwicklungsumgebung weitgehend frei. Da Go standardmäßig nur über den Go Language Server Protocol (gopls)[12] unterstützt wird, kamen primär Editoren mit entsprechender LSP-Unterstützung zum Einsatz. Die bevorzugten Tools waren hier **Visual Studio Code** sowie der moderne Editor **Zed**[24], welcher sich durch seine Performance und klare Nutzeroberfläche auszeichnet. Beide Umgebungen bieten durch die LSP-Anbindung komfortable Features wie Autovervollständigung, Syntaxhervorhebung und Code-Navigation.

Durch diese vielseitige Toolauswahl konnten alle Projektbeteiligten in einer für sie optimalen Umgebung arbeiten, ohne auf zentrale Entwicklungsfeatures verzichten zu müssen. Dies förderte eine produktive Arbeitsweise und erhöhte die Effizienz im Entwicklungsalltag.

2.3 Virtual Private Server

Für das Hosting der Server-Komponenten der **Rocket App** wird ein **Virtual Private Server (VPS)** von **Oracle Cloud** genutzt. Auf diesem Server laufen sowohl das **Backend** als auch die dazugehörige **PostgreSQL-Datenbank**[19]. Um die Dienste zuverlässig und portabel zu betreiben, werden beide Komponenten in separaten **Docker-Containern**[2] ausgeführt.

Die Orchestrierung dieser Container erfolgt über eine **Docker Compose**-Konfiguration, die das Starten, Stoppen und Verwalten der Services vereinfacht. Zusätzlich kommt das Tool **Watchtower** zum Einsatz, das regelmäßig nach aktualisierten Docker-Images prüft. Sobald ein neues Image – beispielsweise des Backends – auf **Docker Hub**[3] verfügbar ist, lädt Watchtower es automatisch herunter und startet den entsprechenden Container neu. Dadurch wird sichergestellt, dass der Server stets mit der aktuellsten Version läuft, ohne manuelles Eingreifen.

2.4 Deployment

Das Deployment des Backends ist vollständig automatisiert und in den Entwicklungsworkflow über **GitHub-Actions**[9] integriert. Bei jedem **Pull Request auf den `master`-Branch**, der

Änderungen im Verzeichnis `rocket-backend` betrifft, wird eine entsprechende GitHub-Action ausgelöst.

Diese Action übernimmt das Erstellen eines neuen Docker-Images auf Basis des aktuellen Codes und lädt es anschließend in ein zentrales **Docker Hub**-Repository hoch. Sobald das neue Image dort verfügbar ist, erkennt **Watchtower** auf dem Oracle-Server das Update und sorgt automatisch für einen nahtlosen Rollout der neuen Backend-Version.

Ein Ausschnitt aus dem verwendeten Deployment-Workflow sieht wie folgt aus:

Listing 1: GitHub Action zur automatisierten Bereitstellung

```
name: Build and Push Docker Image

on:
  push:
    branches:
      - master
    paths:
      - 'rocket-backend/**'

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PAT }}

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Build and Push Docker Image
        uses: docker/build-push-action@v4
        with:
          context: ./rocket-backend
          push: true
          tags: ${{ secrets.DOCKER_USERNAME }}/rocket-backend:latest
```

Durch diese Pipeline wird das Deployment effizient, reproduzierbar und sicher gestaltet – ein entscheidender Vorteil bei der iterativen Weiterentwicklung der Rocket App.

2.5 Setup und Lokale Ausführung

Um das System lokal vollständig auszuführen, werden folgende Tools benötigt:

- **Flutter** – zum Kompilieren und Starten der mobilen Anwendung
- **Docker & Docker Compose** – für das Starten des Backends und der PostgreSQL-Datenbank

2.5.1 1. Backend und Datenbank starten

Zunächst muss das Backend gemeinsam mit der Datenbank gestartet werden. Dazu navigiert man ins Verzeichnis `rocket-backend` und führt dort den folgenden Befehl aus:

```
docker-compose up --build
```

Dieser Befehl setzt voraus, dass im selben Verzeichnis eine `.env`-Datei vorhanden ist. Diese Datei enthält alle Konfigurationsvariablen für Backend und Datenbank. Ein Beispiel-Template sieht wie folgt aus:

```
PORT=8080
APP_ENV=local
BLUEPRINT_DB_HOST=postgres
BLUEPRINT_DB_PORT=5432
BLUEPRINT_DB_DATABASE=blueprint
BLUEPRINT_DB_USERNAME=melkey
BLUEPRINT_DB_PASSWORD=password1234
BLUEPRINT_DB_SCHEMA=public

# generated with openssl rand -base64 64
JWT_SECRET=
API_KEY=

PGADMIN_DEFAULT_EMAIL=admin@admin.com
PGADMIN_DEFAULT_PASSWORD=admin
```

Nach dem Start des Containers wird die Datenbank automatisch durch ein spezielles Migration-Image `migrate/migrate`[14] mit den benötigten Tabellen und Constraints versorgt.

2.5.2 2. Lokale Kommunikation mit dem Handy aktivieren

Damit die mobile App während der lokalen Entwicklung mit dem Backend kommunizieren kann, ist es entscheidend, dass das Handy im selben Netzwerk wie der Entwicklungsrechner ist. Zudem muss entweder:

- die lokale Firewall deaktiviert oder entsprechend konfiguriert werden,
- oder eine Regel erstellt werden, die eingehenden Traffic auf den Port `8080` (bzw. den in der `.env` definierten Port) erlaubt.

2.5.3 3. Mobile App konfigurieren und starten

In der mobilen Anwendung (Verzeichnis `mobile_app`) muss ebenfalls eine `.env`-Datei angelegt werden. Diese enthält die IP-Adresse des Rechners, auf dem das Backend läuft. Die Datei besteht aus nur einer Zeile:

```
BACKEND_URL=http://<IP_DEINES_RECHNERS>:8080
```

Diese IP-Adresse kann je nach System mit dem Befehl `ip a` (Linux) oder `ipconfig` (Windows) ermittelt werden. Diese Angabe ist essenziell, da die mobile App sonst nicht weiß, wohin HTTP-Anfragen gesendet werden sollen. In der Produktionsumgebung ist diese Variable fest definiert, bei der lokalen Entwicklung muss sie jedoch manuell angepasst werden.

Alternativ kann auch die Adresse des Oracle-Servers verwendet werden, sofern dieser verfügbar ist. In diesem Fall lautet die Backend-URL: `https://rocket-app.social`. Dies bietet eine einfache Möglichkeit, auf eine zentral gehostete Instanz zuzugreifen, ohne lokale Anpassungen vornehmen zu müssen.

2.5.4 Hinweis zur mobilen Ausführung

Die App sollte idealerweise auf einem physischen Smartphone installiert und ausgeführt werden. Grund hierfür ist, dass für einige Kernfunktionen (z. B. Geodaten, Schrittzählerdaten) reale Sensorsignale erforderlich sind, welche von Emulatoren in der Regel nicht bereitgestellt werden.

3 Architektur

3.1 VPS Architektur mit HTTPS

Die Abbildung 1 zeigt die Systemarchitektur im produktiven Einsatz, wie sie auf einem Virtual Private Server (VPS) umgesetzt ist. Die Architektur wurde so entworfen, dass sie sicher, skalierbar und wartbar ist. Sie umfasst mehrere Technologien und Container, die gezielt für ihre jeweiligen Stärken eingesetzt werden.

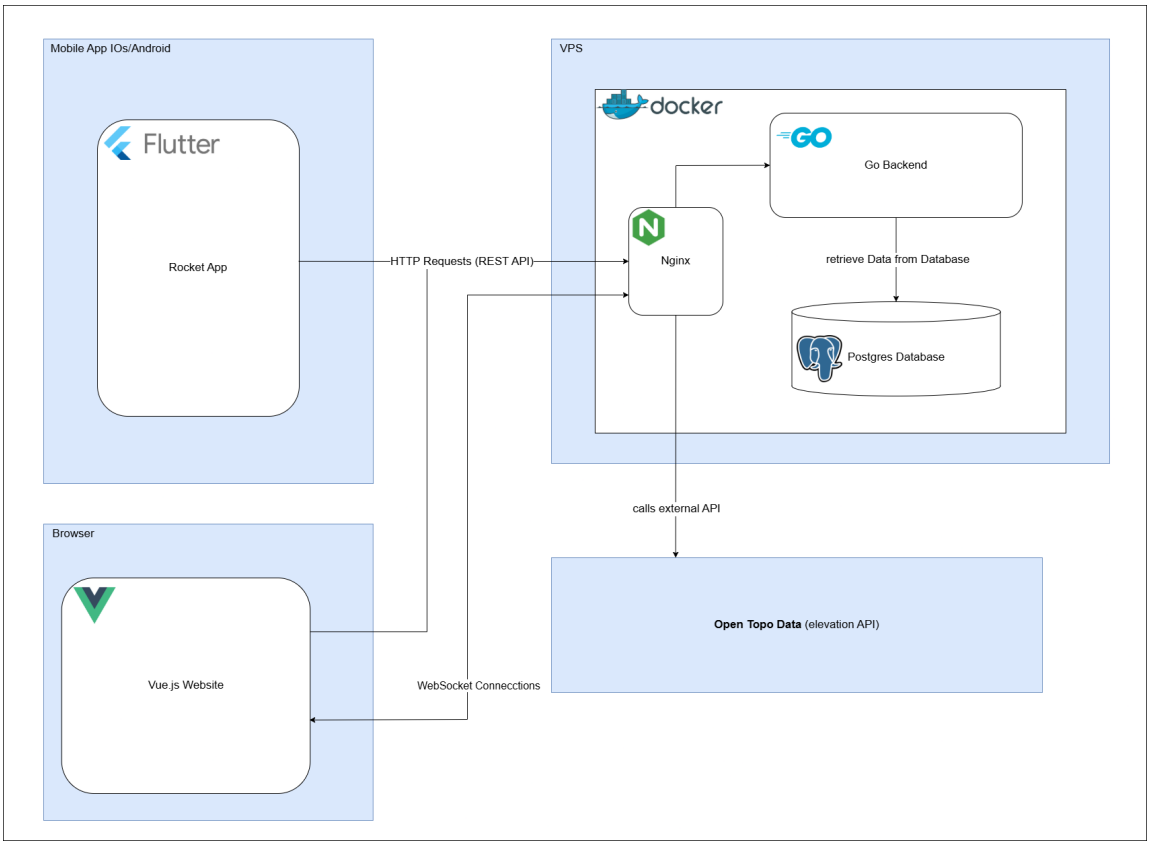

Figure 1: Produktivarchitektur mit Docker, Nginx und HTTPS

Flutter Mobile App Die **Rocket App**, entwickelt mit dem Flutter Framework, läuft auf iOS und Android. Sie dient als Hauptschnittstelle für die Nutzer. Flutter wurde gewählt, weil es eine performante, plattformübergreifende Entwicklung mit einer Codebasis ermöglicht und gleichzeitig native App-Erlebnisse bietet.

Die App sendet ihre Daten über REST-API-Aufrufe direkt an den VPS. Dabei handelt es sich um typische Aktionen wie das Hochladen von Läufen, Synchronisieren von Schritten, Highscorelisten oder das Abrufen von Challenges.

VPS und Containerisierung Der gesamte produktive Stack läuft auf einem **VPS (Virtual Private Server)**. Dies bietet volle Kontrolle über die Serverumgebung bei gleichzeitig moderaten Kosten.

Im VPS sind alle Backend-Komponenten mithilfe von **Docker** containerisiert. Docker erlaubt es, die Applikation isoliert, portabel und versionssicher zu betreiben. Dies erleichtert auch das Deployment (z. B. durch GitHub Actions[9]) sowie die Wartung im laufenden Betrieb.

Nginx als Reverse Proxy Als erste Instanz innerhalb des VPS fungiert **Nginx**[15]. Dieser Reverse Proxy nimmt eingehende HTTP(S)-Anfragen entgegen, kümmert sich um SSL/TLS-Verschlüsselung (z. B. mit Let's Encrypt) und leitet die Anfragen an den Go-Backend-Container weiter.

Die Verwendung von Nginx bringt mehrere Vorteile:

- Trennung von HTTPS-Terminierung und Backend-Logik
- Unterstützung von statischen Dateien und Caching
- Flexible Weiterleitung und Lastverteilung

Go Backend Das **Go-Backend** ist der Kern der Serverlogik. Es verarbeitet alle Anfragen der Mobile App über REST-Schnittstellen. Go wurde gewählt wegen seiner hervorragenden Performance, statischen Typisierung, geringen Laufzeitanforderungen und der guten Eignung für API-Services.

Typische Funktionen des Go-Backends sind:

- Verarbeiten und Speichern von Läufen, Schritten, Chats
- Authentifizierung und Benutzerverwaltung
- Bereitstellen von Geo- und Statistikdaten

PostgreSQL Datenbank Alle persistenten Daten werden in einer **PostgreSQL-Datenbank** gespeichert. PostgreSQL wurde aufgrund seiner Zuverlässigkeit, SQL-Kompatibilität und Unterstützung von Geodaten (PostGIS) ausgewählt. Es läuft ebenfalls als Docker-Container innerhalb des VPS-Netzwerks.

Weitere Komponenten (kurz) Die **Vue.js-Webseite**[22], ebenfalls im Bild dargestellt, kommuniziert wie die App mit dem Backend. Zusätzlich nutzt sie WebSockets für Live-Interaktionen. Eine externe Schnittstelle – hier die **OpenTopoData API**[16] – wird vom Backend genutzt, um Höhendaten für Strecken zu ermitteln.

Zusammenfassung der Architekturvorteile

- **Flutter:** Plattformübergreifende Entwicklung mit nativem Look & Feel
- **Go:** Hochperformant, ideal für APIs
- **Docker:** Portabilität, einfache Updates und Isolierung
- **Nginx:** Reverse Proxy für Sicherheit und Routing
- **PostgreSQL:** Robuste, erweiterbare SQL-Datenbank mit Geo-Support

3.2 Lokale Entwicklungsarchitektur

3.2.1 Unterschied zur Produktionsumgebung

Die lokale Entwicklungsumgebung ist eine vereinfachte Version der Produktivarchitektur. Ziel ist es, einzelne Komponenten unabhängig testen und entwickeln zu können, ohne direkt ein Deployment auf dem VPS durchführen zu müssen.

Im Gegensatz zur vollständigen Produktionsarchitektur (siehe Abbildung 1), bei der mehrere Docker-Container über einen Nginx-Reverse-Proxy orchestriert werden, wird in der lokalen Umgebung lediglich das Backend mit der zugehörigen Datenbank als **lokaler Docker-Container** betrieben.

3.2.2 Architektur der lokalen Umgebung

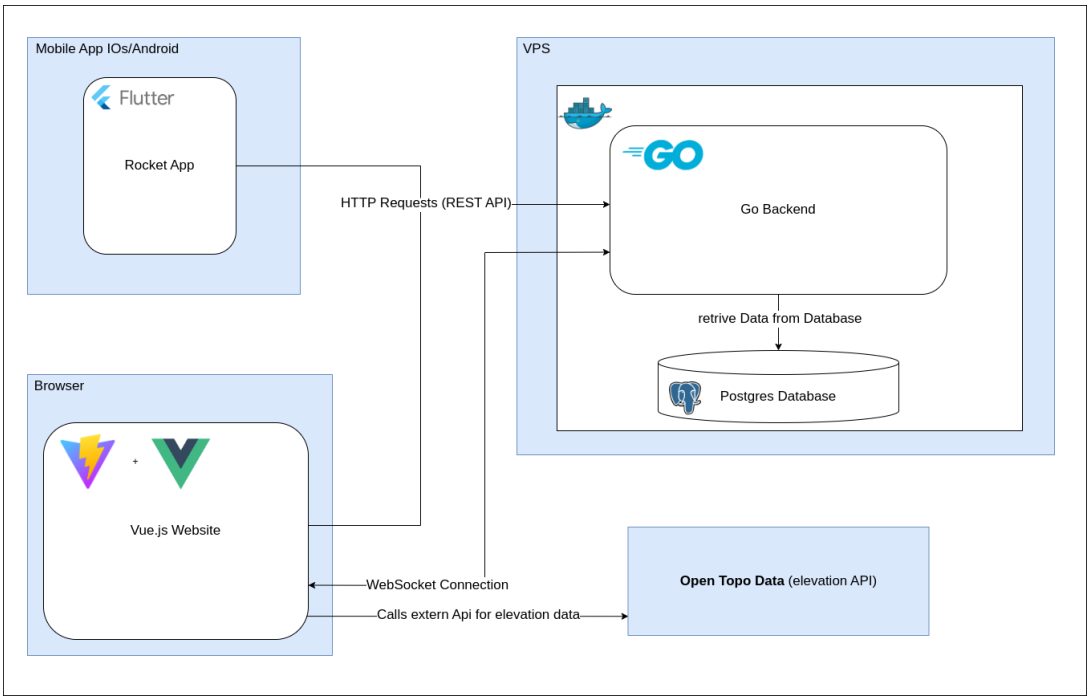

Figure 2: Lokale Entwicklungsumgebung

Backend & Datenbank Das Go-Backend wird lokal in einem Docker-Container gestartet, ebenso wie eine PostgreSQL-Datenbank mit PostGIS-Erweiterung. Dadurch wird eine konsistente Entwicklungsumgebung geschaffen, die der produktiven Struktur sehr nahekommt, jedoch ohne HTTPS-Absicherung oder externe Abhängigkeiten wie Nginx.

Mobile App (Flutter) Die Mobile App kann über USB oder WLAN direkt auf einem realen Gerät installiert werden. Die Verwendung eines echten Geräts ist zwingend notwendig, da die App auf Geo-Koordinaten (GPS) und Schrittzählerdaten zugreift, welche über Gerätesensoren bereitgestellt werden. Emulatoren liefern keine oder nur ungenaue Sensorwerte und sind daher für Entwicklung und Tests ungeeignet. Für das Schrittzählen wird das Plugin `pedometer`[18] verwendet.

Web-Frontend (Vue.js + Vite) Das Web-Frontend basiert auf Vue.js und kann ohne Docker lokal gestartet werden. Dazu wird der **Vite Development Server**[21] verwendet. Dieser bietet schnelles Hot Reloading und ist leichtgewichtig. Eine wichtige Besonderheit ist die

Verwendung eines **Proxy-Setups**, das API-Anfragen aus dem Browser zur lokal laufenden Go-API weiterleitet. Dadurch können Frontend und Backend unabhängig voneinander entwickelt werden, ohne auf ein Deployment angewiesen zu sein.

3.3 Backend Testing

Für das Go-Backend existiert eine umfassende Teststrategie, die auf **Integrationstests** basiert. Ziel ist es, nicht nur die einzelnen Funktionen isoliert zu testen, sondern auch das Zusammenspiel zwischen REST-Endpunkten und Datenbankabfragen realistisch zu überprüfen.

Verwendete Tools Zum Schreiben und Ausführen der Tests kommen zwei populäre Go-Testbibliotheken zum Einsatz:

- `github.com/onsi/ginkgo/v2`[8] – Framework für Behavior-Driven Development (BDD)
- `github.com/onsi/gomega`[10] – Assertion-Bibliothek für lesbare und präzise Testausdrücke

Isolierte Testumgebung mit Testcontainers Zur Laufzeit der Tests wird mit Hilfe von `testcontainers-go`[20] eine isolierte Datenbankinstanz in einem temporären Docker-Container gestartet. Diese Testdatenbank wird automatisch erstellt, mit den notwendigen Migrationsskripten versehen und nach jedem Testlauf vollständig bereinigt.

Dadurch ist sichergestellt, dass:

- Jeder Test in einer identischen, kontrollierten Umgebung läuft
- Seiteneffekte zwischen Tests ausgeschlossen sind
- Produktivdaten niemals verwendet oder überschrieben werden

Testabdeckung Getestet werden:

- Alle REST-Endpunkte des Backends
- Validierung und Fehlerfälle
- Alle relevanten Datenbankoperationen (CRUD)

Diese Teststrategie ermöglicht es, Änderungen im Backend schnell und sicher zu überprüfen, ohne dass manuelles Testen notwendig ist. Sie dient auch als wichtige Grundlage für zukünftige CI/CD-Pipelines.

Automatisierte Testausführung bei Pull Requests Alle Tests werden automatisch ausgeführt, sobald ein Pull Request (PR) auf den `master`-Branch erstellt wird. Dies erfolgt über eine GitHub Action, die das Test-Backend in einer isolierten Umgebung ausführt. Ein PR kann nur gemerged werden, wenn alle Tests erfolgreich durchlaufen. Dies garantiert, dass:

- keine fehlerhaften Änderungen in die Hauptentwicklungslinie gelangen,
- alle Funktionen weiterhin wie erwartet funktionieren,
- die Softwarequalität über alle Sprints hinweg erhalten bleibt.

4 ER-Modell und Datenbankarchitektur

Die Anwendung verwendet **PostgreSQL** als relationale Datenbank, ergänzt durch zwei wichtige Erweiterungen:

- **PostGIS**: Ermöglicht die Speicherung und Verarbeitung von Geodaten, insbesondere Routeninformationen als `LINESTRING`. Diese werden für Laufstrecken und geplante Routen benötigt.
- **pgcrypto**: Wird zur Generierung von `UUIDs` (Universally Unique Identifiers) genutzt, die als Primärschlüssel in fast allen Tabellen verwendet werden.

Die Entscheidung für UUIDs anstelle klassischer Integer-IDs beruht auf mehreren Vorteilen:

- **Sicherheit**: UUIDs sind schwer zu erraten und dadurch weniger anfällig für gezielte Angriffe über ID-Inkremente.
- **Skalierbarkeit**: Sie ermöglichen das Erstellen von IDs über verschiedene Systeme hinweg, ohne Kollisionen befürchten zu müssen.
- **Unabhängigkeit von Kontexten**: Da UUIDs global eindeutig sind, kann etwa eine Laufstrecke unabhängig vom Nutzer eindeutig referenziert werden.

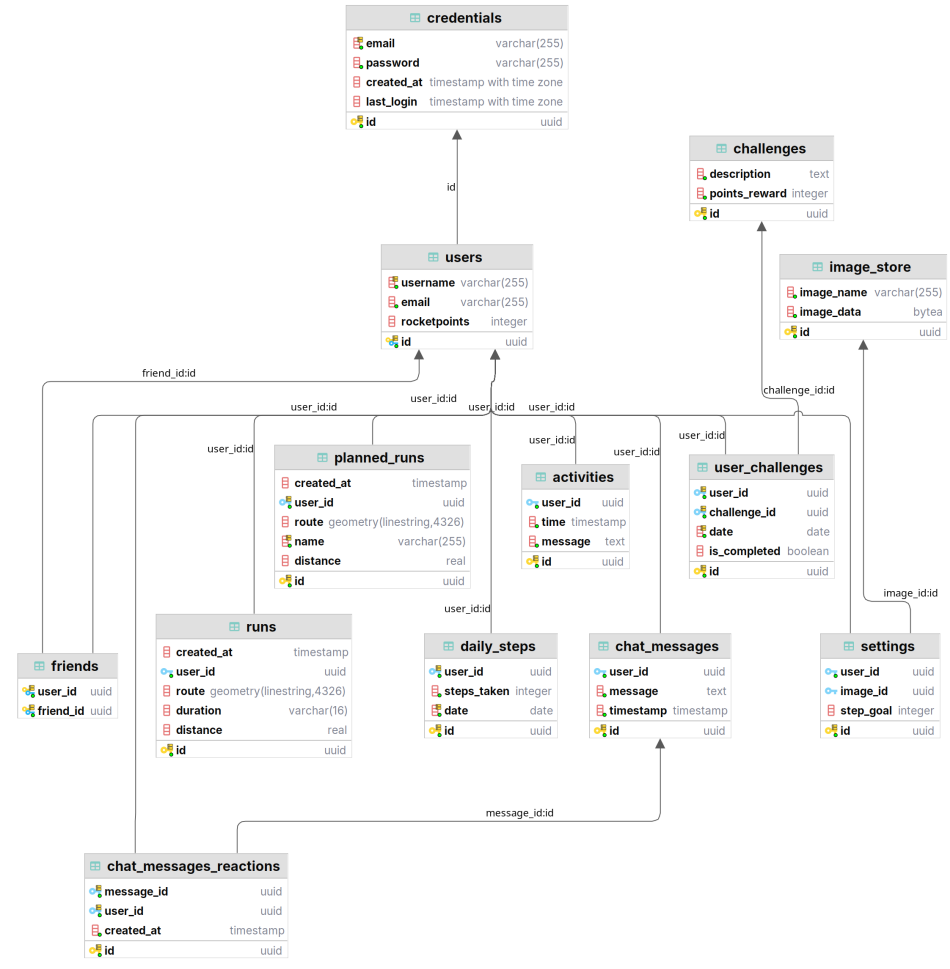

Figure 3: ER-Modell der Rocket-Anwendung

- **users** – Zentrale Entität für Nutzer:innen der App. Enthält Username, Email und den Punktestand (Rocketpoints).
- **credentials** – Separat gespeicherte Zugangsdaten (E-Mail, Passwort) zur besseren Trennung von Authentifizierungs- und Nutzungsdaten.
- **planned_runs** – Beinhaltet vom Nutzer vorgeplante Routen (mit Geometrie), Name und Ziel-Distanz. Essentiell für die Trainingsplanung.
- **runs** – Tatsächlich durchgeführte Läufe, ebenfalls mit Geodaten, Distanz und Dauer. Grundlage für Fortschrittsverfolgung.
- **daily_steps** – Aggregierte tägliche Schrittzahlen pro Nutzer, oft durch Sensorschnittstellen (z. B. Mobilgerät) geliefert.
- **activities** – Logbuch-Funktion für allgemeine Nutzeraktionen. Enthält Textnachrichten und Zeitstempel.
- **friends** – Bidirektionale Freundschaften zwischen Nutzer:innen. Erlaubt soziales Tracking und Interaktion.
- **challenges** – Vorgedefinierte Herausforderungen mit Beschreibung und Punktebelohnung.
- **user_challenges** – Relationstabelle zwischen Nutzer:innen und Herausforderungen, inkl. Statusinformationen wie `is_completed`.
- **settings** – Benutzerbezogene Konfigurationen, insbesondere Zielwerte wie Schrittanzahl oder Profilbild.
- **image_store** – Speicherung binärer Bilddaten (z. B. Avatare, Challenge-Bilder). Gekoppelt an andere Tabellen über `image_id`.
- **chat_messages** – Ermöglicht einfache Nachrichten zwischen Nutzer:innen. Repräsentiert den sozialen Aspekt der App.
- **chat_messages_reactions** – Erweiterung für Reaktionen (z. B. Emojis) auf Chatnachrichten. Enthält Zeitstempel und Verweis auf Nutzer:in.

Alle Relationen sind über `uuid`-Fremdschlüssel miteinander verknüpft, was eine klare logische Trennung und Erweiterbarkeit des Systems unterstützt. Die Kombination aus Geodaten, sozialen Funktionen und gamifizierter Nutzerinteraktion bildet das Rückgrat der Applikation.

5 Authentifizierung und User Validierung

Für die Authentifizierung und Validierung der Nutzer setzen wir auf JSON Web Tokens (JWT) [13]. JWT ermöglicht eine sichere und effiziente Methode, um Benutzeridentitäten zwischen Client und Server zu verifizieren, ohne bei jeder Anfrage die Datenbank abfragen zu müssen.

Der große Vorteil von JWT liegt darin, dass alle nötigen Informationen in einem signierten Token gebündelt sind, das sowohl Integrität als auch Authentizität gewährleistet. Dies reduziert die Serverlast und ermöglicht gleichzeitig eine skalierbare und stateless-Authentifizierung.

In unserer Architektur unterscheiden wir die Handhabung des Tokens zwischen App und Web:

- **Mobile App:** Hier wird der JWT nach erfolgreichem Login sicher im `SecureStorage` gespeichert, um ihn bei nachfolgenden API-Anfragen im Header mitzuschicken. Das Plugin `flutter_secure_storage`[7] gewährleistet, dass der Token verschlüsselt und vor unbefugtem Zugriff geschützt ist. Ein Beispiel zur Speicherung sieht folgendermaßen aus:

Listing 2: Speichern des JWT im Secure Storage

```
await _storage.write(key: 'jwt_token', value: loginResponse.token);
```

- **Webanwendung:** Im Browser wird der JWT als HTTP-Cookie verwaltet. Dies ermöglicht eine einfache automatische Übertragung bei Anfragen und reduziert die Notwendigkeit, den Token manuell im Header zu setzen.

Alle API-Endpunkte außer `/login` und `/register` sind als *protected routes* konfiguriert. Das bedeutet, dass jede Anfrage an diese Endpunkte nur mit einem gültigen JWT im Header akzeptiert wird. Auf diese Weise verhindern wir, dass unautorisierte Nutzer auf geschützte Daten zugreifen können.

Beim Fehlen eines gültigen Tokens oder bei einem abgelaufenen Token wird der Zugriff verweigert und ein entsprechender Fehler (z.B. HTTP 401 Unauthorized) zurückgegeben. Dieses Vorgehen stellt sicher, dass sensible Daten und Funktionen nur authentifizierten Benutzern zugänglich sind.

Der Authentifizierungs-Workflow ist in Abbildung 11 schematisch dargestellt.

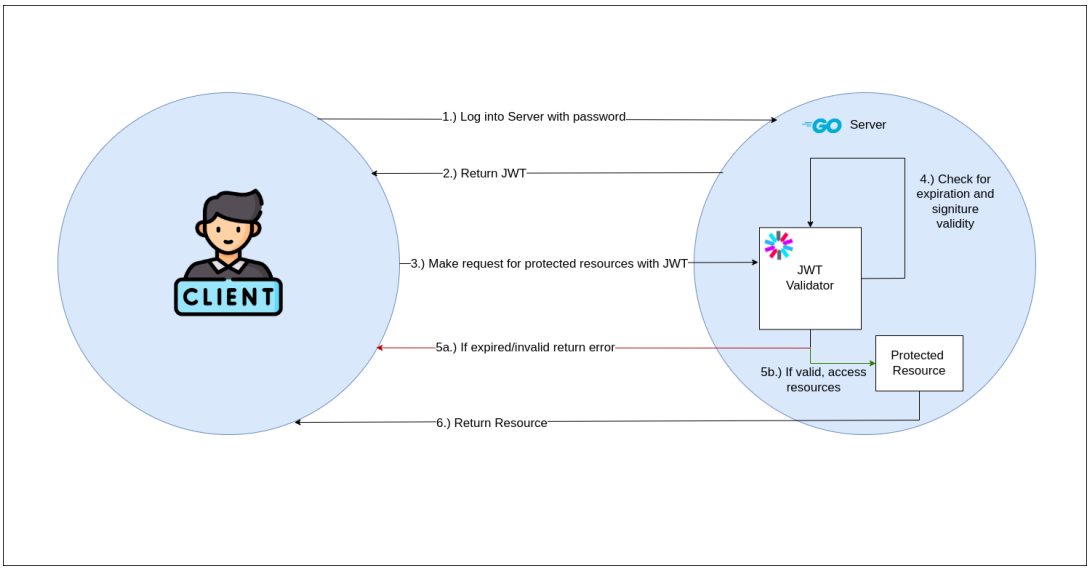

Figure 4: Workflow der Nutzer-Validierung mittels JWT

Die Entscheidung für JWT fiel aufgrund der folgenden Gründe:

- **Skalierbarkeit:** Da JWT stateless ist, muss der Server keine Sitzungsdaten speichern, was die Skalierung der Anwendung erleichtert.
- **Sicherheit:** Durch die Signatur des Tokens wird sichergestellt, dass der Token nicht manipuliert wurde.
- **Flexibilität:** JWT kann problemlos in verschiedenen Client-Typen (Mobile, Web) genutzt werden und erlaubt verschiedene Nutzlasten (Claims).

Durch diese Implementierung gewährleisten wir eine sichere, performante und flexible Authentifizierung, die sich nahtlos in unsere Cross-Plattform-Lösung integriert.

6 Backend

6.1 Endpunkte

Im Folgenden werden alle Backend-Endpunkte der Rocket App API dokumentiert. Die API ist unter dem Prefix `/api/v1` verfügbar. Für alle `/protected`-Routen ist eine Authentifizierung erforderlich (JWT im Header). Der Header für fast alle protected Endpunkte sieht so aus:
Headers:

- `Authorization: Bearer <token>`
- `Content-Type: application/json`

6.1.1 Authentication

POST `/api/v1/register`
Registriert einen neuen Nutzer.

Request Body:

```
{
  "email": "user@example.com",
  "username": "username",
  "password": "securepassword"
}
```

POST `/api/v1/login`
Loggt einen Nutzer ein und gibt ein JWT zurück.

Request Body:

```
{
  "email": "user@example.com",
  "password": "securepassword"
}
```

Response Body:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Hinweis:

- Der JWT wird sowohl im Response-Body zurückgegeben als auch als `HttpOnly`, `Secure`-Cookie mit dem Namen `jwt_token` gesetzt.
- Die Lebensdauer des Cookies beträgt 72 Stunden.
- Das Cookie ist nur über HTTPS übertragbar und nicht über JavaScript auslesbar.

POST `/api/v1/logout`
Entfernt das JWT als Cookie vom Nutzer.

GET `/api/v1/protected/`
Überprüfe, ob ein User angemeldet ist und einen validen JWT besitzt.

6.1.2 User-Endpunkte

GET `/api/v1/user`
Gibt die Informationen des aktuell eingeloggten Nutzers zurück.

Response Body:

```
{
  "id": "UUID",
  "username": "exampleuser",
  "email": "user@example.com",
  "rocket_points": 120
}
```

DELETE `/api/v1/user`
Löscht den aktuell eingeloggten Nutzer.

GET `/api/v1/user/:name`
Gibt die Informationen zu einem Nutzer anhand des Usernames zurück (inkl. Bilddaten, falls vorhanden).

Response Body:

```
{
  "id": "UUID",
  "username": "johndoe",
  "email": "john@example.com",
  "rocket_points": 100,
  "image_name": "profile.png",
  "image_data": "Base64EncodedData"
}
```

POST `/api/v1/user/statistics`
Gibt die Schrittstatistik eines Nutzers für die letzten 7 Tage zurück. Wenn kein Body angegeben ist, wird der eingeloggte Nutzer verwendet.

Request Body (optional):

```
{
  "id": "UUID"
}
```

Response Body:

```
[
  {
    "day": "2025-06-24",
    "steps": 4321
  },
  ...
  {
    "day": "2025-06-18",
    "steps": 6200
  }
]
```

POST `/api/v1/user/image`
Lädt das Profilbild eines Nutzers (eigener oder über ID im Body).

Request Body (optional):

```
{
  "user_id": "UUID"
}
```

Response Body:

```
{
  "username": "johndoe",
  "name": "profile.jpg",
  "mime_type": "image/jpeg",
  "data": "Base64EncodedImage"
}
```

GET `/api/v1/user/rocketpoints`
Gibt die RocketPoints des eingeloggten Nutzers zurück.

Response Body:

```
{
  "rocket_points": 125
}
```

GET `/api/v1/users`
Gibt eine Liste aller Nutzer mit ihren Daten und Bildern zurück.

Response Body:

```
[
  {
    "id": "UUID",
    "username": "johndoe",
    "email": "john@example.com",
    "rocket_points": 100,
    "steps": 2345,
    "image_name": "profile.jpg",
    "image_data": "Base64EncodedImage"
  },
  ...
]
```

POST `/api/v1/protected/updateSteps`
Aktualisiert die Schrittanzahl des aktuellen Nutzers.

Request Body:

```
{
  "steps": 1234
}
```

6.1.3 Settings

GET `/api/v1/protected/settings`
Gibt aktuelle Benutzereinstellungen zurück.

POST `/api/v1/protected/settings/step-goal`
Aktualisiert das Schrittziel des Nutzers.

Request Body:

```
{
  "stepGoal": 8000
}
```

POST `/api/v1/protected/settings/image`
Aktualisiert das Profilbild eines authentifizierten Benutzers.

Headers:

- `Authorization: Bearer <token>`
- `Content-Type: multipart/form-data`

Form-Data Felder:

- `image` – Bilddatei (JPEG, PNG etc.)

Beispielanfrage mit `curl`:

```
curl -X POST http://localhost:8080/api/v1/protected/settings/
    image \
  -H "Authorization: Bearer <your_jwt>" \
  -F "image=@profile.jpg"
```

DELETE `/api/v1/protected/settings/image`
Löscht das aktuelle Profilbild.

POST `/api/v1/protected/settings/userinfo`
Aktualisiert die Benutzerinformationen wie Name, E-Mail oder Passwort. Es können beliebige Felder gesendet werden – sie sind alle optional. Die Passwortänderung erfordert das aktuelle Passwort.

Request Body:

```
{
  "name": "Max␣Mustermann",
  "email": "max@example.com",
  "currentPassword": "oldpass123",
  "newPassword": "newpass456"
}
```

6.1.4 Challenges

GET `/api/v1/protected/challenges/new`
Gibt die aktuellen täglichen Challenges zurück.

POST `/api/v1/protected/challenges/complete`
Markiert eine Challenge als abgeschlossen.

Request Body:

```
{
  "challenge_id": "uuid-abc...",
  "rocket_points": 20
}
```

GET `/api/v1/protected/challenges/progress`
Gibt den Fortschritt der täglichen Challenges zurück.

POST `/api/v1/protected/challenges/invite`
Lädt einen Freund zu einer Challenge ein.

Request Body:

```
{
  "challenge_id": "ccf4fe0e-002f-4fbd-80d6-0670db8a979b",
  "friend_id": "80c2f290-7644-4f76-9d87-cb5d39501c42"
}
```

6.1.5 Ranking

GET `/api/v1/protected/ranking/users`
Gibt eine Liste der Top 100 Nutzer nach Rocket Points zurück (globales Ranking).

Response Body:

```
[
  {
    "id": "9cfa8b97-7adf-4d2b-b408-882db5adf511",
    "username": "john_doe",
    "email": "john@example.com",
    "rocketPoints": 320,
    "imageName": "avatar.png",
    "imageData": "<base64-encoded>",
    "steps": 0
  },
  ...
]
```

GET `/api/v1/protected/ranking/friends`
Gibt das Ranking der Freunde des Nutzers nach Rocket Points zurück.

Response Body:

```
[
  {
    "id": "8ab7f178-e0e3-4cfc-a8b2-20b420b8f2ee",
    "username": "jane_doe",
    "email": "jane@example.com",
    "rocketPoints": 250,
    "imageName": "jane.png",
    "imageData": "<base64-encoded>",
    "steps": 0
  },
  ...
]
```

6.1.6 Friends

GET `/api/v1/protected/friends`
Gibt eine Liste aller Freunde des angemeldeten Nutzers mit Bild, RocketPoints und aktuellen Schritten zurück.

Response Body:

```
[
  {
    "id": "user-uuid-1",
```

```
    "username": "alice",
    "email": "alice@example.com",
    "rocketPoints": 230,
    "imageName": "avatar.png",
    "imageData": "<base64-encoded>",
    "steps": 5374
  },
  ...
]
```

POST `/api/v1/protected/friends/add`
Fügt einen neuen Freund anhand des Benutzernamens hinzu.

Request Body:

```
{
  "friend_name": "bob"
}
```

DELETE `/api/v1/protected/friends/:name`
Entfernt einen Freund anhand des Benutzernamens.

GET `/api/v1/protected/following/:id`
Gibt eine Liste aller Nutzer zurück, denen der Benutzer mit der angegebenen UUID folgt.

Response Body:

```
[
  {
    "id": "user-uuid-2",
    "username": "charlie",
    "email": "charlie@example.com",
    "rocketPoints": 180,
    "imageName": "charlie.png",
    "imageData": "<base64-encoded>",
    "steps": 0
  },
  ...
]
```

GET `/api/v1/protected/followers/:id`
Gibt eine Liste aller Nutzer zurück, die dem Benutzer mit der angegebenen UUID folgen.

Response Body:

```
[
  {
    "id": "user-uuid-3",
    "username": "david",
    "email": "david@example.com",
    "rocketPoints": 160,
    "imageName": "david.png",
    "imageData": "<base64-encoded>",
    "steps": 0
  },
  ...
]
```

6.1.7 Runs

POST `/api/v1/protected/runs`
Lädt eine neue Laufaktivität hoch.

Request Body:

```
{
  "route": "LINESTRING(....)",
  "duration": "25m30s",
  "distance": 5.23
}
```

GET `/api/v1/protected/runs`
Gibt alle hochgeladenen Läufe des authentifizierten Nutzers zurück.

Response Body (200 OK):

```
[
  {
    "id": "run-uuid-1",
    "route": "LINESTRING(....)",,
    "duration": "25m30s",
    "distance": 5.23
  },
  ...
]
```

DELETE `/api/v1/protected/runs/:id`
Löscht den Lauf mit der angegebenen UUID.

POST `/api/v1/protected/runs/plan`
Plant einen zukünftigen Lauf.

Request Body:

```
{
  "route": "LINESTRING(....)",
  "name": "Morning␣Run",
  "distance": 7.5
}
```

GET `/api/v1/protected/runs/plan`
Gibt alle geplanten Läufe des authentifizierten Nutzers zurück.

Response Body (200 OK):

```
[
  {
    "id": "planned-run-uuid-1",
    "route": "LINESTRING(....)",
    "name": "Morning␣Run",
    "distance": 7.5
  },
  ...
]
```

DELETE `/api/v1/protected/runs/plan/:id`
Löscht den geplanten Lauf mit der angegebenen UUID.

6.1.8 Aktivitäten und Chat

GET `/api/v1/protected/activities`
Gibt Aktivitäten des aktuellen Benutzers sowie seiner Freunde zurück. Die Antwort enthält optional Benutzerbilder.

- **Response:**

```
{
  "username": "johndoe",
  "activities": [
    {
      "name": "johndoe",
      "time": "2025-06-25T13:14:00Z",
      "message": "Completed a 5.00 km run in 28 minutes",
      "imageName": "profile.jpg",
      "imageType": "image/jpeg",
      "imageData": "base64-encoded-data"
    },
    ...
  ]
}
```

GET `/api/v1/protected/chat/history`
Gibt alle bisherigen Chatnachrichten des Benutzers zurück.

- **Response:**

```
{
  "messages": [
    {
      "id": "uuid",
      "username": "alice",
      "message": "Hello, everyone!",
      "timestamp": "2025-06-25T12:30:00Z",
      "reactions": 3,
      "hasReacted": true
    },
    ...
  ]
}
```

GET `/api/v1/protected/ws/chat`
Öffnet eine WebSocket-Verbindung für den globalen Chat.

- **WebSocket-Nachrichtenformat (Text, JSON):**

```
{
  "username": "bob",
  "message": "Let's go for a run!",
  "timestamp": "2025-06-25T13:40:00Z"
}
```

- Reaktionen und Nutzerverbindungen werden serverseitig verwaltet. Nachrichten werden über einen `ChatHub` verteilt.

6.2 Docker Container

Die gesamte Anwendung besteht aus mehreren Docker-Containern, die über ein gemeinsames Netzwerk `blueprint` verbunden sind. Das Setup basiert auf `docker-compose` und ist für den produktiven Einsatz auf einem VPS optimiert. Alle internen Services sind sauber voneinander getrennt und verwenden eigene Images.

backend `zephiron/rocket-backend:latest`
Dieser Container stellt die REST-API der Anwendung bereit. Er ist in Go implementiert und verbindet sich mit der PostgreSQL-Datenbank. Die wichtigsten Umgebungsvariablen beinhalten z. B. `JWT_SECRET` und `DATABASE_URL`. Das Image wird vorab gebaut und auf Docker Hub veröffentlicht, um die Build-Zeit auf dem Server zu sparen.

frontend `zephiron/rocket-website:latest`
Die statische Website mit Vue.js läuft in einem eigenen Container und wird ebenfalls über Docker Hub bereitgestellt. Auch hier wird durch die vorgefertigten Images eine schnelle und ressourcensparende Bereitstellung ermöglicht.

postgres `postgis/postgis:latest`
Die Datenbank verwendet PostgreSQL mit der PostGIS-Erweiterung für mögliche geographische Erweiterungen. Persistenz wird über ein Docker-Volume gewährleistet. Ein Healthcheck stellt sicher, dass der Container erst als *ready* markiert wird, wenn Verbindungen angenommen werden können.

migrate `migrate/migrate`
Dieser Container führt automatisch Migrationen beim Start aus. Er nutzt das offizielle `migrate` CLI-Tool und greift über ein Volume auf die lokalen Migrationsdateien zu.

reverse-proxy `nginx:1.27-alpine`
Dieser NGINX-Container agiert als Reverse-Proxy für alle eingehenden HTTP(S)-Anfragen. Er leitet:

- `/api/v1/` an das **Backend** weiter (inkl. WebSocket-Unterstützung)
- `/elevation-api/` an den externen Service `https://api.opentopodata.org`
- alle anderen Pfade an das **Frontend**

Besonders hervorzuheben ist, dass der Reverse-Proxy **HTTPS über Let's Encrypt-Zertifikate** unterstützt. Alle Anfragen auf Port 80 werden automatisch auf HTTPS (Port 443) umgeleitet. Die Zertifikate werden lokal im Container gemountet:

```
ssl_certificate /etc/letsencrypt/live/rocket-app.social/fullchain
    .pem;
ssl_certificate_key /etc/letsencrypt/live/rocket-app.social/
    privkey.pem;
```

Dadurch ist sichergestellt, dass die gesamte Anwendung über eine verschlüsselte Verbindung (HTTPS) erreichbar ist — ein wichtiger Sicherheitsaspekt im Produktivbetrieb.

Vorteil der Build-Strategie: Da die `backend`- und `frontend`-Images bereits lokal entwickelt und anschließend auf Docker Hub (`zephiron/*`) veröffentlicht werden, muss der VPS keine eigenen Build-Prozesse ausführen. Dies spart sowohl Speicher als auch CPU-Zeit und minimiert mögliche Build-Fehler während der Bereitstellung. Die Container können dadurch direkt per `docker pull` geladen und gestartet werden.

7 Handy App

7.1 Foreground Task

Die App verwendet das Flutter-Plugin `flutter_foreground_task`[5], um dauerhaft im Vordergrund bzw. Hintergrund Schrittzahlen zu erfassen und regelmäßig an das Backend zu senden. Dies ist vor allem unter Android notwendig, da das Betriebssystem Hintergrundprozesse stark einschränkt und andernfalls wichtige Trackingdaten verloren gehen könnten.

Die Foreground Task besteht aus zwei getrennten Instanzen, welche in unterschiedlichen Isolates (Flutter-Threads) laufen. Diese kommunizieren nicht direkt über Speicher, sondern über Ports:

- **Hauptanwendung (Main Isolate)**: Dies ist der Hauptthread, in dem auch die Benutzeroberfläche läuft. Dort wird der Foreground-Task initialisiert und gestartet.

Listing 3: Initialisierung des Foreground Task

```
FlutterForegroundTask.initCommunicationPort();
_initService();
await _startService();
```

Listing 4: Registrierung des Task Handlers

```
@pragma('vm:entry-point')
void startCallback() {
  FlutterForegroundTask.setTaskHandler(MyTaskHandler());
}
```

- **Hintergrundprozess (TaskHandler Isolate)**: Die Klasse `MyTaskHandler` implementiert den Hintergrunddienst. Dieser läuft in einem separaten Isolate, das keine direkte Verbindung zur Hauptanwendung hat. Die wichtigsten Aufgaben des TaskHandlers sind:
  - Schrittzählung mit dem Plugin `pedometer`
  - Tageswechsel erkennen und Schritte zurücksetzen
  - Regelmäßige Synchronisierung mit dem Backend alle 15 Minuten

Listing 5: Schrittzählung starten

```
_sub = Pedometer.stepCountStream.listen(_onStepCount);
```

Listing 6: Tageswechsel erkennen

```
if (!_isSameDay(now, _lastResetDate)) {
  _baselineSteps = event.steps;
  _stepsToday = 0;
}
```

Listing 7: Synchronisierung mit dem Backend

```
await api.DailyStepsApi.sendDailySteps(steps, jwt);
```

Kommunikation zwischen Instanzen Da Haupt- und Hintergrundprozess getrennt sind, wird der JWT über eine Nachricht übertragen und im SharedPreferences-Store abgelegt. Ebenso werden Schrittzahlen regelmäßig an die Hauptinstanz gemeldet. Dies geschieht über:

Listing 8: Schritte an Hauptprozess senden

```
FlutterForegroundTask.sendDataToMain(_stepsToday);
```

Es ist wichtig zu verstehen, dass diese Trennung notwendig ist, aber auch eine gewisse Komplexität mit sich bringt – z.B. bei der Persistierung von Zuständen oder dem Teilen von Authentifizierungsinformationen.

Berechtigungen Damit der Foreground Task korrekt ausgeführt werden kann, sind verschiedene Berechtigungen notwendig:

- Notification-Permission
- Aktivitätserkennung (für Schrittzählung)
- Optional: Deaktivieren von Battery-Optimierung (vorbereitet, aber auskommentiert)

Listing 9: Abfrage der Berechtigungen

```
if (Platform.isAndroid) {
  if (await Permission.activityRecognition.isDenied) {
    await Permission.activityRecognition.request();
  }
}
```

Dieser Foreground Task bildet die Grundlage für die dauerhafte Aktivitätsaufzeichnung und ermöglicht eine robuste Synchronisation mit dem Backend – unabhängig davon, ob sich die App im Vorder- oder Hintergrund befindet.

7.2 Willkommens-, Login- und Registrierungsseite

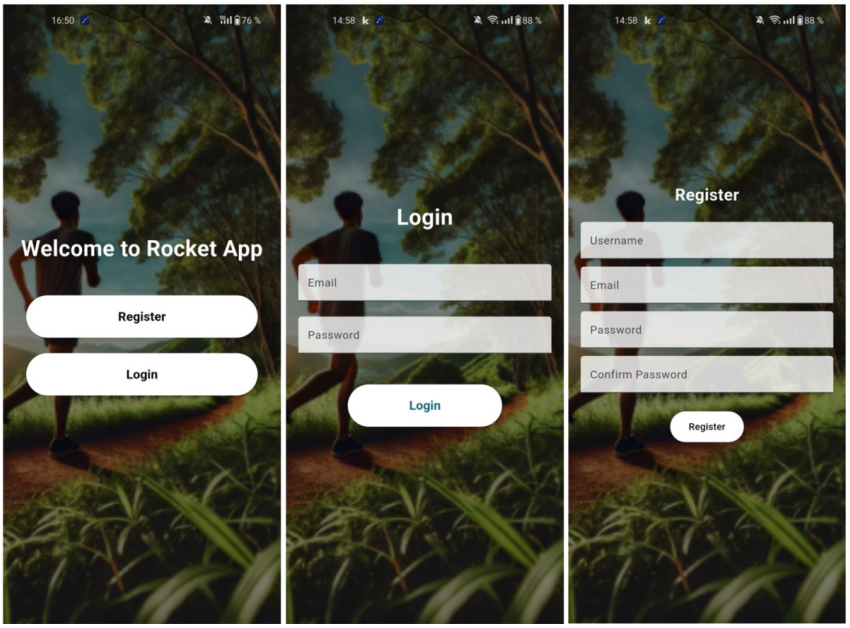

Figure 5: Startbildschirm der Rocket App mit Anmelde- und Registrierungsoptionen: Nutzer können sich mit E-Mail und Passwort einloggen oder ein neues Konto erstellen, um mit dem Tracking zu beginnen.

Nach dem Start der App wird der Benutzer auf die **WelcomePage** geführt, die als zentrales Einstiegstor zur Anwendung dient. Dort kann zwischen den beiden Kernaktionen *Login* und *Registrierung* gewählt werden. Das Hintergrundbild erzeugt dabei eine motivierende visuelle Atmosphäre. Die Buttons führen jeweils zu den entsprechenden Seiten.

Navigation und Aufbau Die Navigation erfolgt über den `Navigator`-Stack von Flutter. Die Buttons sind als `ElevatedButton` umgesetzt und führen beim Tippen zur jeweiligen Seite:

Listing 10: Navigation zu Register- oder Login-Seite

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => RegisterPage()),
);
```

Registrierungsseite Die **RegisterPage** enthält ein validiertes Formular mit Feldern für Benutzername, E-Mail, Passwort und Passwortbestätigung. Das Formular wird durch ein `GlobalKey<FormState>` gesteuert. Die Validierung stellt sicher, dass z. B. die E-Mail-Adresse korrekt formatiert ist und die Passwörter übereinstimmen.

Bei erfolgreicher Validierung wird die Registrierungsanfrage über eine asynchrone Methode an das Backend gesendet:

Listing 11: Registrierungsfunktion mit Backend-Aufruf

```
final registerResponse = await RegisterApi.register(
  _emailController.text,
  _usernameController.text,
  _passwordController.text,
);
```

Nach erfolgreicher Registrierung wird der Nutzer automatisch zur Login-Seite weitergeleitet. Fehler, wie doppelte Benutzer oder ungültige Daten, werden über eine Snackbar angezeigt.

Loginseite Die **LoginPage** erlaubt registrierten Nutzern die Authentifizierung mittels E-Mail und Passwort. Auch hier wird ein `Form`-Widget verwendet, um Eingabefelder zu validieren. Im Erfolgsfall wird das JWT des Nutzers über die Login-API bezogen und sicher mit `flutter_secure_storage` gespeichert:

Listing 12: Speichern des JWT nach Login

```
await _storage.write(key: 'jwt_token', value: loginResponse.token);
```

Da die App mit einem Foreground-Service zur Hintergrundsynchronisation arbeitet, wird das Token außerdem über `FlutterForegroundTask.sendDataToTask` an den Hintergrundprozess übergeben.

Listing 13: Token an Hintergrundprozess senden

```
FlutterForegroundTask.sendDataToTask({'jwt_token': loginResponse.token
    });
```

Im Anschluss erfolgt die Weiterleitung auf die Hauptnavigation der App:

Listing 14: Navigation zum App-Hauptmenü

```
Navigator.pushReplacement(
  context,
  MaterialPageRoute(builder: (context) => AppNavigator(title: 'Rocket␣
      App')),
);
```

Fehlermeldungen Bei Eingabefehlern oder Backend-Fehlern werden visuelle Rückmeldungen gegeben – entweder direkt im Formular oder per Snackbar. So wird sichergestellt, dass der Benutzer bei jedem Schritt eine klare Rückmeldung erhält.

Zusammenfassung Die Einstiegspunkte der App sind klar strukturiert und nutzen bewährte Flutter-Pattern für Navigation, Formularverarbeitung und Zustandshandhabung. Das Zusammenspiel aus UI, Validierung und API-Kommunikation sorgt für eine nahtlose User Experience bereits beim ersten Kontakt mit der App.

7.3 Startseite

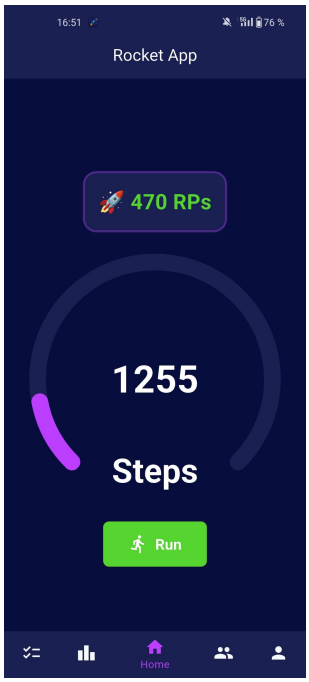

Figure 6: Startseite der Rocket App mit aktueller Schrittanzahl, verdienten Rocket Points (RPs) und „Run“-Button zum Aufrufen der Tracking-Seite.

Die zentrale Startansicht der App ist die **RunPage**, die dem Nutzer eine Übersicht über seine täglichen Schritte, das aktuelle Schrittziel sowie die gesammelten Rocket Points (RPs) bietet. Über den Button *Run* wird man direkt auf die Tracking-Seite weitergeleitet.

Datenanbindung Die RunPage lädt beim Start die Benutzerdaten über zwei asynchrone API-Aufrufe:

- `RocketPointsApi.fetchRocketPoints(jwt)` holt die aktuellen RPs.
- `SettingsApi.getSettings(jwt)` liefert das individuell gesetzte Schrittziel.

Die Werte werden anschließend im State gespeichert:

Listing 15: Speichern von RP und Schrittziel

```
setState(() {
  rocketPoints = response.rocketPoints;
  dailyGoal = settings.stepGoal;
});
```

Schrittfortschritt Die Komponente `StepCounterWidget` visualisiert den Fortschritt mit einem benutzerdefinierten Circular Painter. Die aktuelle Schrittanzahl wird dabei durch einen Foreground-Task regelmäßig aktualisiert. Der Fortschrittswert ergibt sich aus:

$$\text{progress} = \frac{\text{currentSteps}}{\text{dailyGoal}}$$

Listing 16: Progressberechnung im StepCounterWidget

```
double progress = currentSteps / dailyGoal;
```

Rocket Points Anzeige Die gesammelten Rocket Points werden in einer auffälligen Karte mit Raketen-Emoji angezeigt. Diese Anzeige reagiert dynamisch auf Lade- und Fehlerzustände:

```
rocketPoints != null
  ? '$rocketPoints␣RPs'
  : '...␣RPs'
```

Navigation über BottomNavigationBar Die gesamte App wird über eine eigene Navigationskomponente namens `CustomMenuBar` gesteuert. Diese ist als `BottomNavigationBar` mit fünf Einträgen aufgebaut:

- Challenges
- Leaderboard
- Home (zentral hervorgehoben)
- Friends
- Profile

Der Startindex der Navigation ist standardmäßig auf `2` gesetzt, womit beim App-Start automatisch die RunPage (Home) angezeigt wird.

Listing 17: Initiale Seitenauswahl im AppNavigator

```
const AppNavigator({
  required this.title,
  this.initialIndex = 2, // Home
});
```

Tracking starten Durch Betätigen des zentralen *Run*-Buttons wird die Tracking-Seite aufgerufen. Die Navigation erfolgt über einen `MaterialPageRoute`:

Listing 18: Navigation zur Tracking-Seite

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => TrackingPage(title: 'Tracking
      ')),
);
```

Zusammenfassung Die Startseite dient als tägliches Dashboard für Nutzeraktivitäten. Durch Kombination aus API-Anbindung, Live-Schrittzählung und einfacher Bedienung bildet sie den funktionalen Mittelpunkt der Rocket App.

7.4 Runs

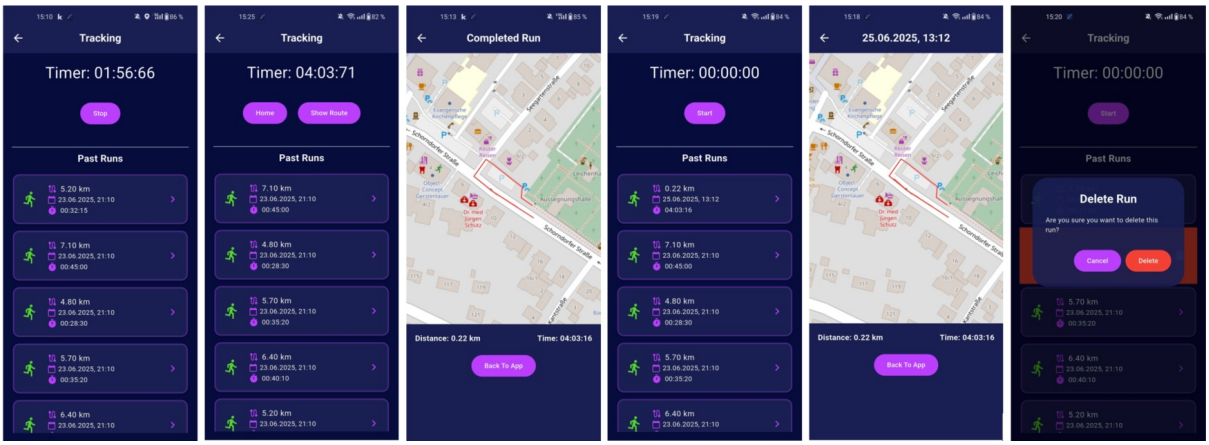

Figure 7: Tracking-Bereich der Rocket App: Übersicht über vergangene Läufe mit Distanz, Zeit und Karte der Laufroute; inklusive Timer- und Tracking-Funktion, Routendarstellung und Option zum Löschen einzelner Einträge.

In unserer App wird die Lauf-Route eines Nutzers kontinuierlich mithilfe von GPS-Daten erfasst. Diese Positionsdaten werden als *GeoPoints* gespeichert, um später die gesamte Laufstrecke rekonstruieren und visualisieren zu können.

7.4.1 Verwendung von OpenStreetMap

Für die Darstellung der Routen auf der Karte setzen wir die Open-Source-Kartenplattform *OpenStreetMap* (OSM)[17] ein. Die Flutter-Bibliothek `flutter_osm_plugin`[6] ermöglicht die nahtlose Integration der OSM-Karten in die App und bietet umfangreiche Funktionen, wie z.B. das Zeichnen von Wegen, die Anzeige von Markern und Nutzerpositionen.

OSM bietet den Vorteil, dass die Kartendaten frei verfügbar und anpassbar sind. So können wir die Laufstrecke direkt auf der Karte abbilden, indem wir die gesammelten Geo-Koordinaten als Pfad über die Karte legen.

7.4.2 Erfassung und Speicherung der GPS-Daten

Während eines Laufs werden fortlaufend die aktuellen GPS-Koordinaten des Nutzers abgefragt. Diese Koordinaten werden in einer Liste von `GeoPoint` Objekten gesammelt, die jeweils die `latitude` und `longitude` enthalten.

Listing 19: Hinzufügen eines GeoPoints zur Route

```
_geoPoints.add(GeoPoint(
  latitude: position.latitude,
  longitude: position.longitude,
));
```

Diese Liste bildet die gesamte Laufstrecke ab. Neben den Geo-Koordinaten wird die verstrichene Zeit mit einem Timer gemessen und später zusammen mit der Route gespeichert.

7.4.3 Senden der Route an das Backend

Um die Laufdaten dauerhaft zu speichern, werden die Geo-Punkte in ein standardisiertes Format, den sogenannten `LINESTRING`, umgewandelt. Dieses Format wird häufig in geografischen Informationssystemen (GIS) verwendet, um Linien oder Wege zu repräsentieren.

Listing 20: Erzeugen eines LINESTRINGs

```
String lineString = 'LINESTRING(' +
  _geoPoints.map((p) => '${p.longitude} ${p.latitude}').join(', ') +
  ')';
```

Die Umwandlung ist notwendig, damit das Backend die Route als eine durchgehende Linie interpretieren kann. Anschließend wird der `LINESTRING` zusammen mit der Laufzeit an die API gesendet:

Listing 21: Senden der Laufdaten ans Backend

```
await TrackingApi.addRun(jwt, lineString, duration, distance);
```

7.4.4 Darstellung der Route in der App

Zur Visualisierung der gespeicherten Laufstrecken wird die Route vom Backend geladen und als Liste von GeoPoints in der App dargestellt. Über die OpenStreetMap-Karte wird dann die Strecke mit der Funktion `drawRoad` gezeichnet.

Dabei werden die einzelnen Segmente der Route als Fußwege (RoadType.foot) mit einer roten Linie hervorgehoben:

Listing 22: Zeichnen der Route auf der Karte

```
for (int i = 0; i < _routePoints.length - 1; i++) {
  await _mapController.drawRoad(
    _routePoints[i],
    _routePoints[i + 1],
    roadType: RoadType.foot,
    roadOption: RoadOption(
      roadColor: Colors.red,
      roadWidth: 8,
    ),
  );
}
```

Diese Visualisierung ermöglicht dem Nutzer, die genaue Strecke seines Laufs nachvollziehen zu können. Zusätzlich werden wichtige Informationen wie Gesamtdistanz und Laufzeit übersichtlich angezeigt.

7.4.5 Berechnung der Gesamtdistanz

Die Gesamtdistanz wird aus den GeoPoints berechnet, indem für jedes aufeinanderfolgende Punktpaar die Entfernung mithilfe der Haversine-Formel bestimmt wird. Diese Formel berechnet die kürzeste Entfernung über die Erdoberfläche zwischen zwei Koordinaten.

Listing 23: Berechnung der Entfernung zweier GeoPoints

```
double _calculateDistanceBetween(GeoPoint a, GeoPoint b) {
  const double R = 6371; // Erdradius in Kilometern
  double dLat = _deg2rad(b.latitude - a.latitude);
  double dLon = _deg2rad(b.longitude - a.longitude);
  double lat1 = _deg2rad(a.latitude);
  double lat2 = _deg2rad(b.latitude);

  double hav = sin(dLat / 2) * sin(dLat / 2) +
      sin(dLon / 2) * sin(dLon / 2) * cos(lat1) * cos(lat2);
  double c = 2 * atan2(sqrt(hav), sqrt(1 - hav));
  return R * c;
}
```

Die Summe aller Teilstrecken ergibt die Gesamtstrecke des Laufs.

7.4.6 Verwaltung von Läufen

Alle gespeicherten Läufe werden unterhalb des Timers in einer Liste dargestellt. Jeder Eintrag zeigt:

- die zurückgelegte Distanz,
- das Datum und die Uhrzeit des Laufs,
- sowie die Laufdauer.

Lauf starten und stoppen Ein Lauf kann durch Tippen auf die `Start`-Schaltfläche begonnen werden. Dabei wird ein `Stopwatch` aktiviert, um die Zeit zu messen, und über die `Geolocator`-Bibliothek werden kontinuierlich GPS-Daten aufgezeichnet. Während des Laufens wird die Zeit in Echtzeit oben im Interface angezeigt.

Nach dem Laufen kann der Nutzer den Lauf durch Tippen auf `Stop` beenden. Anschließend erscheint eine neue Auswahl: Die Route kann angezeigt (`Show Route`) oder zur Startseite zurückgekehrt werden (`Home`).

Listing 24: Starten des Trackings

```
setState(() {
  _isTracking = true;
  _geoPoints.clear();
  _stopwatch.reset();
  _stopwatch.start();
});
```

Listing 25: Stoppen des Trackings und Speichern

```
_stopwatch.stop();
_positionStream?.cancel();
await TrackingApi.addRun(jwt, lineString, duration, distance);
```

Einsehen eines gespeicherten Laufs Durch Tippen auf einen Listeneintrag in der Historie wird die zugehörige Laufroute auf einer Karte visualisiert. Hierzu wird die gespeicherte `LINESTRING`-Route zurück in `GeoPoints` konvertiert und mit der Funktion `drawRoad` auf der OpenStreetMap-Karte gezeichnet. Zusätzlich werden Distanz und Dauer erneut eingeblendet.

Listing 26: Navigation zur Detailansicht einer Route

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => RoutePage(
      title: _formatDate(run.createdAt),
      routePoints: parseLineString(run.route),
      elapsedTime: run.duration,
    ),
  ),
);
```

Löschen eines Laufs Durch ein Wischen (Swipe) eines Lauf-Eintrags nach links wird die Option eingeblendet, den Lauf zu löschen. Bevor die Daten endgültig entfernt werden, erscheint ein Dialog zur Bestätigung der Aktion. Dies verhindert versehentliches Löschen und erhöht die Nutzerfreundlichkeit.

Listing 27: Löschbestätigung per Dialog

```
confirmDismiss: (direction) async {
  return await showDialog(
    context: context,
    builder: (context) => AlertDialog(
      title: Text('Delete Run'),
      content: Text('Are you sure you want to delete this run?'),
      actions: [
        ElevatedButton(onPressed: () => Navigator.of(context).pop(false
            ), child: Text('Cancel')),
        ElevatedButton(onPressed: () => Navigator.of(context).pop(true)
            , child: Text('Delete')),
      ],
    ),
  );
},
```

Listing 28: Entfernen des Laufs aus der Datenbank

```
await TrackingApi.deleteRun(jwt, run.id);
```

Diese Funktionen bilden zusammen eine vollständige Tracking-Suite, mit der Nutzer ihre Aktivitäten genau aufzeichnen, verwalten und visualisieren können.

7.5 Freundesliste

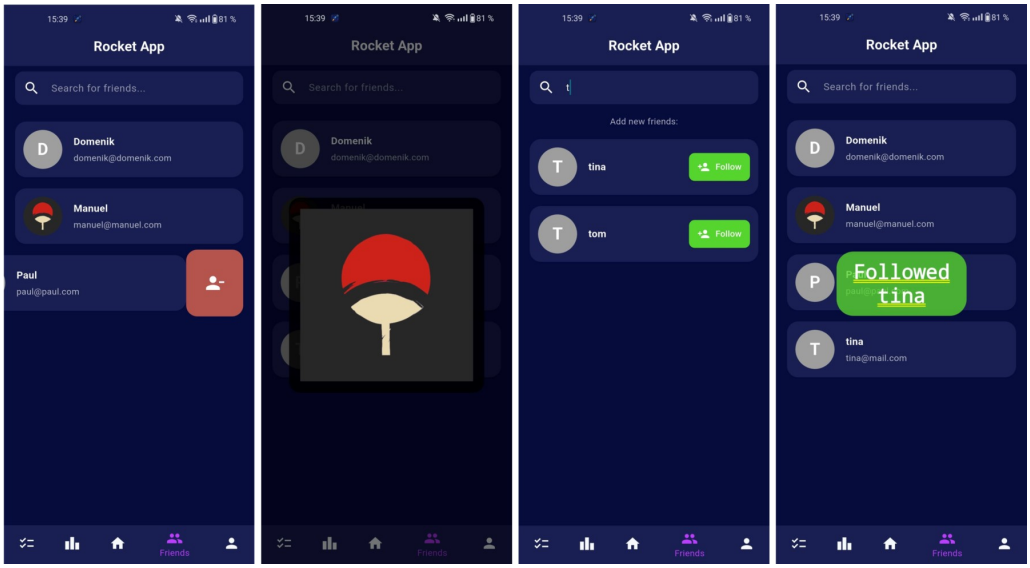

Figure 8: Freundesliste der Rocket App: Nutzer können nach Freunden suchen, Kontakte hinzufügen oder entfernen sowie deren Mailadresse einsehen.

Die Freundesliste ist ein zentraler Bestandteil der sozialen Interaktion innerhalb der Rocket App. Sie erlaubt das Verwalten bestehender Kontakte sowie das gezielte Suchen und Hinzufügen neuer Nutzer. Sie ermöglicht es Nutzer:innen somit, mit anderen Personen in Kontakt zu treten.

Die Funktionalität wird durch serverseitige API-Abfragen in Kombination mit lokalen Filtern ermöglicht.

Suchfunktion und Nutzerliste Oberhalb der Ansicht befindet sich eine Suchleiste, mit der gezielt nach Nutzern anhand ihres Usernamens gesucht werden kann. Die Eingabe filtert dynamisch sowohl:

- bereits gefolgte Freunde (links in der Abbildung) und
- nicht gefolgte Nutzer, die mit einem "`Follow`"-Button angezeigt werden (dritte Ansicht).

Bei jeder Eingabe (über `onChanged`) wird der Zustand `_searchQuery` aktualisiert. Die Liste der bereits gefolgten Freunde sowie möglicher neuer Kontakte wird in der `build()`-Methode auf Basis dieses Werts dynamisch gefiltert. Die Filterung funktioniert über eine **Substring-Suche**: Der eingegebene Suchtext wird mit jedem Benutzernamen (kleinbuchstaben-normalisiert) verglichen. Enthält der Benutzername den Suchtext, wird das Element angezeigt:

Listing 29: Filter-Logik für Freundesliste und Suchergebnisse

```
final filteredFriends = friends.where((f) =>
  _searchQuery.isEmpty || f.username.toLowerCase().contains(
     _searchQuery.toLowerCase())
).toList();

final searchResults = allUsers.where((u) {
  final matchesQuery = _searchQuery.isNotEmpty &&
                       u.username.toLowerCase().contains(_searchQuery.
                          toLowerCase());
  final notFriendYet = !friends.any((f) => f.id == u.id);
  return matchesQuery && notFriendYet;
}).toList();
```

Freunde entfernen per Swipe-Geste Bereits gefolgte Kontakte lassen sich per Swipe-Geste (nach links) über das `flutter_slidable`-Paket entfernen. Es erscheint eine Schaltfläche zum Entfernen des Freundes (`Icons.person_remove`). Ein Tap löst einen API-Aufruf aus, um den Kontakt serverseitig zu löschen. Danach wird die Freundesliste neu geladen.

Listing 30: Unfollow eines Freundes per Slidable

```
await FriendsApi.deleteFriend(jwt, friend.username);
final freshFriends = await FriendsApi.getAllFriends(jwt);
setState(() {
  friends = freshFriends;
});
```

Bestehende Freunde anzeigen und entfernen Bereits gefolgte Nutzer:innen werden als Karten in einer Liste angezeigt. Diese Karten enthalten:

- den Nutzernamen
- die E-Mail-Adresse
- ein Profilbild (oder Initiale, falls kein Bild vorhanden ist)

Sowohl für bestehende als auch neue Kontakte wird das Profilbild angezeigt – sofern verfügbar. Fehlt ein Bild, wird stattdessen der erste Buchstabe des Nutzernamens dargestellt. Bei Klick auf ein Profilbild wird es in einer vergrößerten Ansicht angezeigt.

Listing 31: Profilbildanzeige und Zoom-Dialog

```
GestureDetector(
  onTap: () => _showImageDialog(friend.imageData!),
  child: CircleAvatar(
    backgroundImage: MemoryImage(friend.imageData!),
  ),
);
```

Neue Freunde suchen und hinzufügen Wird ein Benutzername eingegeben, der zu Nutzer:innen passt, die noch nicht gefolgt werden, erscheinen diese unter dem Abschnitt *Add new friends*. Jede Karte zeigt Username und Bild sowie einen grünen "Follow"-Button.

Bei Betätigen dieses Buttons:

1. wird ein `POST`-Request an die API `/api/v1/protected/friends/add` gesendet,
2. die Freundesliste und Userliste wird lokal aktualisiert (`setState`),
3. die Eingabe wird geleert,
4. und ein **Overlay-Popup** mit dem Text `Followed <name>` erscheint für 2 Sekunden.

Listing 32: Freund hinzufügen und Overlay anzeigen

```
await FriendsApi.addFriend(jwt, user.username);
_showFollowOverlay(user.username);
```

Listing 33: Overlay-Feedback nach erfolgreichem Hinzufügen

```
OverlayEntry(
  builder: (context) => Center(
    child: Container(
      child: Text('Followed\n$name'),
    ),
  ),
);
```

Asynchrone Datenbindung und Sicherheit Die gesamte Seite ist ebenfalls durch einen sicheren Authentifizierungstoken (JWT) abgesichert, der lokal über `flutter_secure_storage` ausgelesen wird. Alle Serverzugriffe (z.B. `getAllFriends`, `addFriend`, `deleteFriend`) verwenden diesen Token im Header.

Design und Farben Alle Einträge (Friend-Cards) sind optisch abgesetzt, farblich abwechselnd hinterlegt (dunkelblau und violett) und enthalten:

- Profilbild bzw. Initiale
- Nutzername
- E-Mail-Adresse

Durch die Kombination aus direkter Interaktion, visuellem Feedback und API-Anbindung stellt die Freundesliste ein intuitives und funktionales Element der App dar, das soziale Vernetzung aktiv unterstützt. Durch die sichere, asynchrone Datenbindung und die enge Verzahnung mit dem Server-Backend wird eine stets aktuelle und konsistente Darstellung der sozialen Kontakte innerhalb der App gewährleistet.

7.6 Challenges

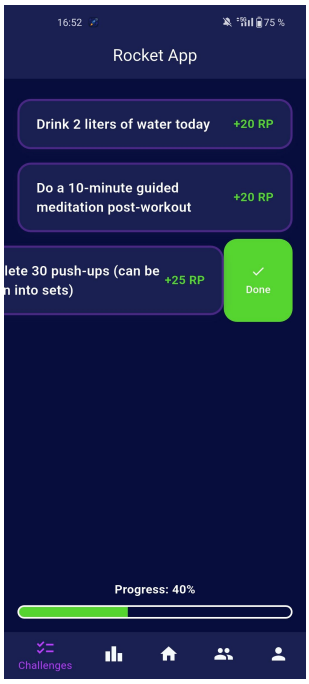

Figure 9: Challenges-Ansicht der Rocket App: Nutzer können tägliche Aufgaben wie Trinken, Meditation oder Push-ups absolvieren und dafür Rocket Points sammeln – inklusive Fortschrittsanzeige in Prozent.

Die Challenge-Seite zeigt eine Auswahl an täglichen Aufgaben, die Nutzer:innen zur Förderung eines gesunden Lebensstils absolvieren können. Jede absolvierte Aufgabe wird mit sogenannten **Rocket Points (RP)** belohnt. Die Seite bietet dabei sowohl eine klare Übersicht über aktuelle Challenges als auch eine visuelle Rückmeldung über Fortschritt und Erfolge.

Laden der Herausforderungen Beim Öffnen der Seite werden zunächst alle aktuellen täglichen Challenges per API geladen:

Listing 34: Laden der Tages-Challenges vom Server

```
final challenges = await ChallengesApi.fetchChallenges(jwt);
setState(() {
  _challenges = challenges;
});
```

Zusätzlich wird der aktuelle Fortschritt abgerufen – also wie viele Challenges bereits erledigt wurden und wie viele insgesamt verfügbar sind:

Listing 35: Abfrage des Challenge-Fortschritts

```
final progress = await ChallengesApi.fetchChallengeProgress(jwt);
_completedChallenges = progress.completed;
_totalChallenges = progress.total == 0 ? 1 : progress.total;
```

Darstellung der Challenges Jede Challenge wird als visuell abgesetzte Karte mit zwei Informationen angezeigt:

- **Beschreibung der Aufgabe** (z. B. “Drink 2 liters of water today”)

- **Belohnung in Rocket Points** (z. B. "+20 RP")

Die Karten besitzen ein modernes Design mit weichem Schatten, abgerundeten Ecken und einem farblich abgesetzten Rand. Die Gestaltung ist vollständig dynamisch und anpassbar über zentrale Farbkonsstanten.

Markieren als erledigt (Swipe to complete) Eine Challenge kann als **„erledigt"** markiert werden, indem die Karte entweder nach links oder rechts gewischt wird – realisiert mit `flutter_slidable`. Dabei wird folgende Logik ausgeführt:

1. Die Challenge wird aus der Liste entfernt
2. Eine API-Anfrage markiert sie als erledigt
3. Die Fortschrittsanzeige wird aktualisiert
4. Eine visuelle Animation mit den gewonnenen Punkten erscheint

Listing 36: Challenge als erledigt markieren

```
await ChallengesApi.markAsDone(jwt, challenge.id, challenge.points);
setState(() {
  _challenges.remove(challenge);
});
```

Visuelles Feedback: RP-Overlay Wird eine Challenge abgeschlossen, erscheint ein animiertes Overlay, das die erhaltenen Punkte anzeigt. Es wird zentral im Bildschirm eingeblendet, bewegt sich nach oben und verblasst anschließend. Dabei kommen Flutter-Animationen und ein `OverlayEntry` zum Einsatz:

Listing 37: Overlay mit Punktanzeige bei Erfolg

```
OverlayEntry(
  builder: (context) => Center(
    child: Text('+25 RP', style: TextStyle(...)),
  ),
);
```

Fortschrittsbalken Am unteren Rand wird der aktuelle Fortschritt durch einen **linearen Balken** dargestellt. Die Prozentzahl basiert auf dem Verhältnis von erledigten zu allen täglichen Challenges. Sowohl der Text "Progress: 40%" als auch der Balken selbst aktualisieren sich automatisch.

Listing 38: Berechnung des Fortschrittswertes

```
double progressValue = (_completedChallenges / _totalChallenges).clamp
    (0.0, 1.0);
```

Der Balken nutzt eine abgerundete Rahmenstruktur mit weißem Rand und ist in leuchtendem Grün gefüllt, sobald Fortschritte erzielt werden.

Ziel und Motivation Die Challenges sind ein integraler Bestandteil der Gamification-Strategie der Rocket App. Sie dienen dazu, Alltagsaufgaben in spielerischer Form mit messbaren Fortschritten zu verknüpfen. Durch visuelle Belohnungen, Swipe-Gesten und Fortschrittsanzeige entsteht ein direkter Anreiz zur täglichen Nutzung.

Die API-Endpunkte, die zur Realisierung dieser Seite genutzt werden, sind unter Abschnitt `/api/v1/protected/challenges` dokumentiert.

7.7 Leaderboard

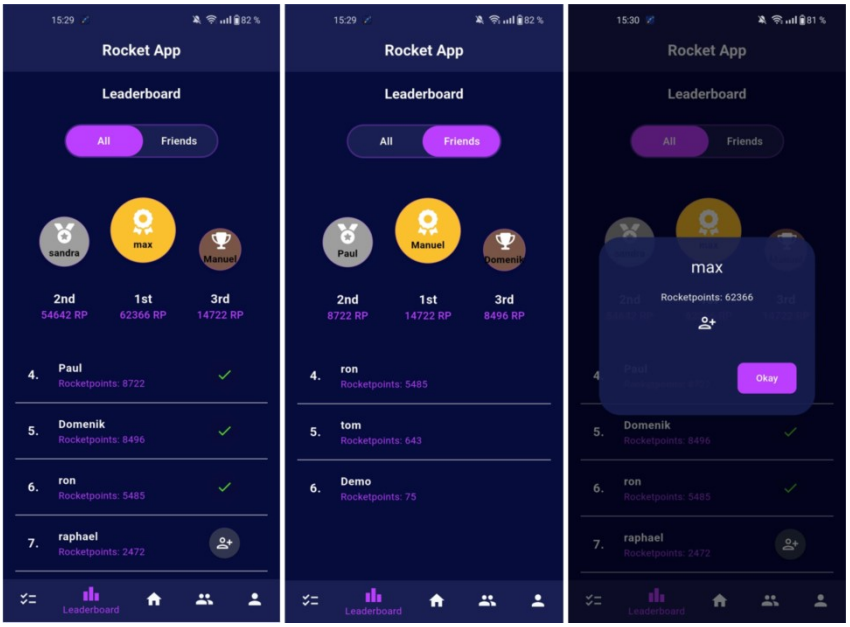

Figure 10: Leaderboard der Rocket App: Anzeige der globalen und freundesbezogenen Rangliste basierend auf gesammelten Rocket Points – mit Detailansicht einzelner Nutzerprofile.

Die Leaderboard-Ansicht visualisiert den aktuellen Punktestand aller Nutzer:innen der Rocket App. Sie dient als soziales Vergleichselement und fördert die Motivation durch Ranglistenmechanismen. Dabei können wahlweise:

- alle Nutzer:innen (*All*) oder
- nur Freund:innen (*Friends*)

angezeigt werden.

Tab-Umschaltung (All vs. Friends) Die Ansicht verwendet einen animierten Umschalter zur Auswahl zwischen den zwei Modi. Die jeweils angezeigte Liste (`displayedUsers`) wird beim Umschalten aktualisiert:

Listing 39: Wechsel zwischen allen Nutzer:innen und Freund:innen

```
void switchTab(int tabIndex) {
  setState(() {
    selectedTab = tabIndex;
    displayedUsers = tabIndex == 0 ? allUsers : friends;
  });
}
```

Platz 1–3: Podestanzeige Die drei führenden Nutzer:innen werden prominent auf einem Podium dargestellt. Jeder Platz (1.–3.) ist farblich und größenmäßig differenziert:

- **Gold** für Platz 1 mit größtem Avatar
- **Silber** für Platz 2
- **Bronze** für Platz 3

Die Avatare beinhalten einen Rang-Icon (Medaille, Award, Pokal) sowie den Namen. Ein Tap auf den Avatar öffnet ein modales Dialogfenster mit weiteren Infos.

Detaildialog mit Freundschaftsoption Wird auf einen Podest-Avatar getippt, erscheint ein modaler Dialog (`AlertDialog`), der Folgendes zeigt:

- Benutzername
- Gesamtpunktzahl
- Option zum Hinzufügen als Freund (sofern noch nicht befreundet)

Listing 40: Dialog zur Freundschaftsanfrage aus dem Podium

```
IconButton(
  icon: Icon(isFriend(user) ? Icons.check : Icons.person_add_alt),
  onPressed: isFriend(user) ? null : () async {
    addFriend(user);
    Navigator.of(context).pop();
  },
);
```

Rangliste ab Platz 4 Alle weiteren Nutzer:innen (ab Platz 4 oder ab Platz 1, wenn ¡3) werden unter dem Podium in einer scrollbaren Liste angezeigt. Jede Zeile enthält:

- Rangnummer
- Benutzername
- Aktueller Punktestand
- Freundschaftsstatus (Icon zum Hinzufügen oder Häkchen)

Ist man bereits mit einem Nutzer befreundet, wird ein grüner Haken angezeigt. Andernfalls erscheint ein “Add Friend”-Icon als Button.

Freund:innen hinzufügen aus der Rangliste Nutzer:innen im “All”-Modus können direkt durch Tippen auf das Icon als Freund hinzugefügt werden. Der Button wird nach erfolgreichem API-Call deaktiviert.

Listing 41: Freundschaft aus der Rangliste anfragen

```
ElevatedButton(
  onPressed: isFriend(user) ? null : () => addFriend(user),
  child: Icon(isFriend(user) ? Icons.check : Icons.person_add_alt),
);
```

Datenquelle und Aktualisierung Die Daten für beide Listen (“All” und “Friends”) werden beim Initialisieren der Seite über einen gesicherten API-Aufruf geladen. Verwendet wird ein lokal gespeicherter JWT:

Listing 42: Laden der Ranglisten über API

```
final fetchedAllUsers = await RankingApi.fetchUserRankings(jwt);
final fetchedFriends = await RankingApi.fetchFriendRankings(jwt);
```

Die Daten bleiben so lange bestehen, bis die Seite verlassen oder manuell aktualisiert wird. Beim Hinzufügen eines neuen Freundes wird die Freundesliste ergänzt.

Funktion und Motivation Das Leaderboard steigert die Nutzerbindung durch Wettbewerb. Die visuelle Podiumsdarstellung, unmittelbare Vergleichbarkeit sowie einfache Interaktionsmöglichkeiten (z. B. neue Freund:innen entdecken) fördern eine nachhaltige und aktive Nutzung der App.

Die zugehörigen Backend-Endpunkte finden sich unter `/api/v1/protected/leaderboard` und `/friends/add`.

7.8 Profil und Einstellungen

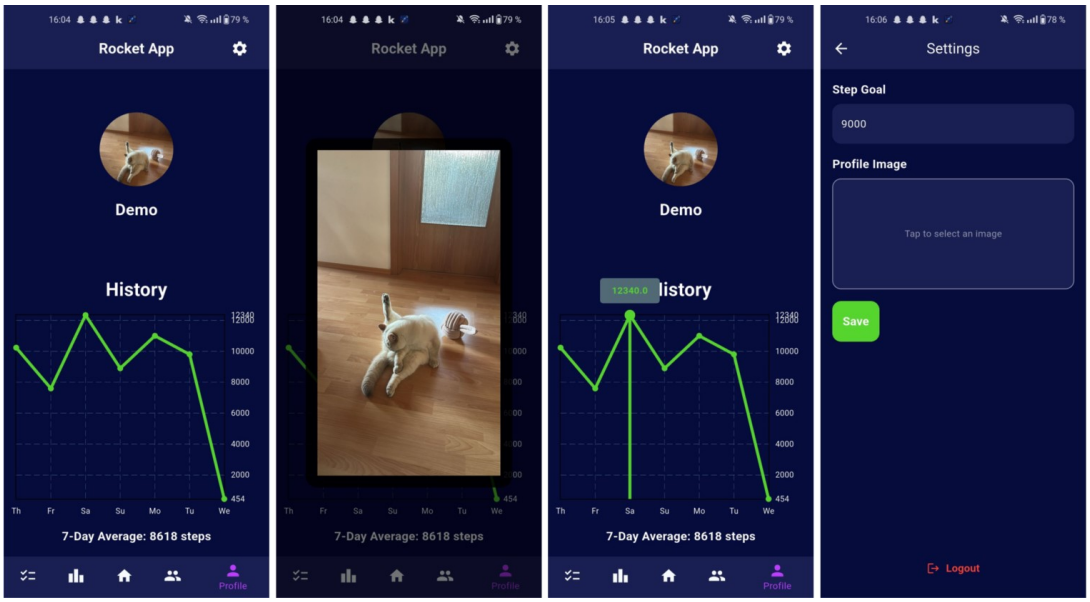

Figure 11: Profil- und Einstellungsbereich der Rocket App: Anzeige der Schritt-Historie mit 7-Tage-Durchschnitt, Möglichkeit zur Änderung des Profilbilds sowie Festlegung eines individuellen Schrittziels

7.8.1 Profilseite

Die Profilseite kombiniert zwei Hauptkomponenten: ein Profilbild mit Nutzernamen (oben) sowie eine visuelle Darstellung der Schrittanzahl der letzten sieben Tage (unten).

Profilbild & Username Das Profilbild wird beim Laden der Seite per API von einem Base64-codierten String in ein Bild umgewandelt und in einem runden Avatar dargestellt. Ist kein Bild vorhanden, wird stattdessen der Anfangsbuchstabe des Benutzernamens gezeigt. Beim Antippen des Bildes wird dieses in einem modalen Vollbilddialog angezeigt:

Listing 43: Bild als Dialog anzeigen

```
GestureDetector(
  onTap: () => _showImageDialog(Uint8List.fromList(imageData)),
  child: CircleAvatar(...),
);
```

Schritthistorie (Line Chart) Darunter befindet sich ein Liniendiagramm, das die tägliche Schrittanzahl der vergangenen sieben Tage visualisiert. Die Datenpunkte werden per API geladen, wobei jeder Punkt einem Wochentag zugeordnet ist.

Die Darstellung nutzt das Paket `fl_chart`. Das Diagramm zeigt:

- Tagesnamen auf der x-Achse
- Schrittanzahl auf der y-Achse (rechte Seite)
- Durchschnittswert als Zahl unter dem Diagramm

Listing 44: Diagrammkonfiguration für Schrittverlauf

```
LineChartBarData(
  spots: List.generate(
    stepsData.length,
    (index) => FlSpot(index.toDouble(), stepsData[index]),
  ),
  isCurved: false,
  color: ColorConstants.greenColor,
  barWidth: 4,
);
```

Beim Antippen eines Tages erscheint oberhalb der Kurve ein Tooltip, der die genaue Schrittanzahl für diesen Tag anzeigt.

7.8.2 Einstellungsseite

Die Einstellungsseite bietet zwei konfigurierbare Optionen:

- Tägliches Schrittziel (numerischer Wert)
- Profilbild (uploadbar aus Galerie)

Schrittziel anpassen Das Ziel kann frei gewählt und gespeichert werden. Eingabefehler wie ungültige Zahlen oder leere Felder werden dabei abgefangen:

Listing 45: Gültigkeitsprüfung Schrittziel

```
if (stepGoal == null || stepGoal <= 0) {
  _errorMessage = 'Please␣enter␣a␣valid␣step␣goal␣greater␣than␣0';
}
```

Profilbild ändern Durch Antippen des Upload-Felds öffnet sich die Bildgalerie. Nach Auswahl wird das Bild in der UI angezeigt und beim “Save”-Klick hochgeladen:

Listing 46: Bild aus Galerie auswählen

```
final pickedFile = await picker.pickImage(source: ImageSource.gallery);
if (pickedFile != null) {
  _selectedImage = File(pickedFile.path);
}
```

Daten speichern Nach Klick auf “Save” werden Schrittziel und Profilbild über zwei separate API-Aufrufe gespeichert. Bei Erfolg erscheinen Snackbar-Benachrichtigungen.

Logout-Funktion Ein “Logout”-Button entfernt das gespeicherte JWT-Token und leitet zur Startseite um:

Listing 47: Abmelden und Navigation zur Welcome-Seite

```
await _storage.delete(key: 'jwt_token');
Navigator.pushAndRemoveUntil(
  context,
  MaterialPageRoute(builder: (context) => const WelcomePage()),
  (route) => false,
);
```

Visuelles und UX-Design Alle Eingabefelder und Container sind durchgängig visuell abgesetzt (rund, mit Rand, einheitliches Padding). Die Farben der App bleiben auch hier konsistent (dunkles Primärlayout, grüne Buttons für Aktionen).

Die API-Endpunkte zur Verwaltung von Profil- und Einstellungselementen befinden sich unter `/api/v1/protected/user` und `/api/v1/protected/settings`.

8 Zeiterfassung

Zu Beginn des Projekts haben wir das Tool **Everhour**[25] zur Zeiterfassung verwendet. Dieses bot eine sehr komfortable Integration mit unserem GitHub-Repository: Arbeitszeiten konnten direkt über Commits und Branches automatisch getrackt werden, was einen präzisen und effizienten Überblick über die aufgewendete Zeit ermöglichte.

Leider war diese Funktionalität nur im Rahmen eines Testzeitraums verfügbar. Nach dessen Ablauf war die Nutzung des Tools in vollem Umfang kostenpflichtig. Aus diesem Grund mussten wir auf eine kostenlose Alternative umsteigen und haben uns für **Clockify**[26] entschieden.

Clockify bietet zwar nicht die Tiefe GitHub-Integration wie Everhour, eignet sich aber hervorragend für eine einfache und manuelle Zeiterfassung. Um die vollständige Dokumentation unserer Projektzeiten zu gewährleisten, haben wir die zuvor mit Everhour erfassten Stunden nachträglich in Clockify eingetragen. Dies erklärt auch den deutlichen Ausschlag im zeitlichen Verlauf der erfassten Stunden, wie er in Abbildung 12 zu erkennen ist.

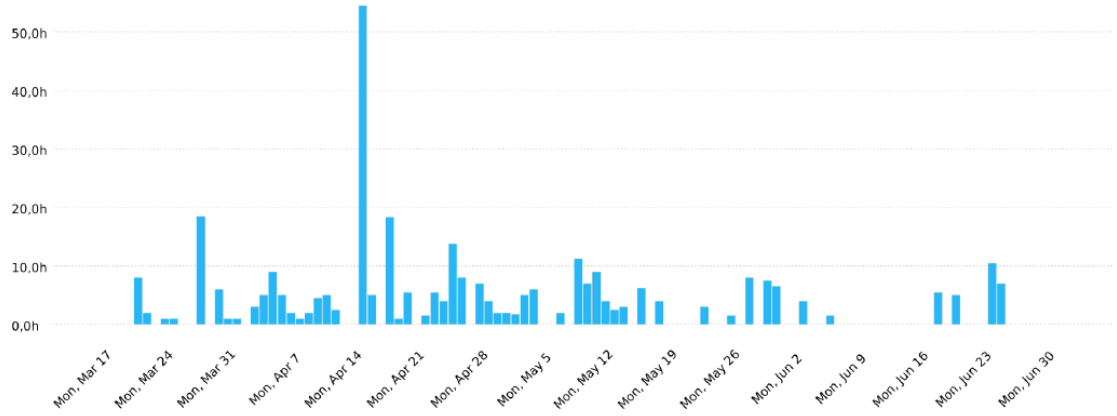

Figure 12: Gesamte Zeiterfassung mit Clockify

Die dargestellten Werte zeigen die insgesamt investierten Stunden, aufgeteilt nach Kalenderwochen. Die Daten umfassen die gesamte Projektlaufzeit und beziehen sich auf alle Teammitglieder gemeinsam. Die Stundenangaben sind jeweils in Stunden summiert.

Im Folgenden ist eine detaillierte Aufschlüsselung der investierten Zeit nach einzelnen Teammitgliedern zu sehen. Diese Darstellung vermittelt einen transparenten Überblick über die individuelle Arbeitsverteilung innerhalb des Teams sowie den Gesamtaufwand.

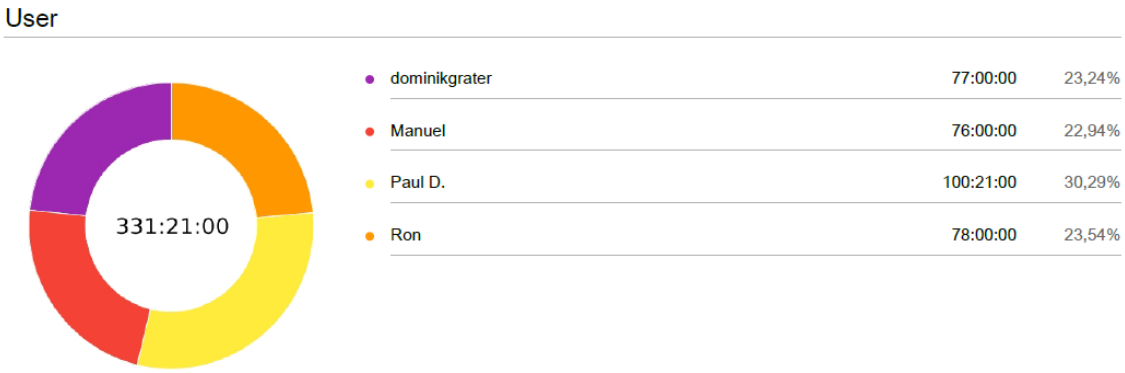

Figure 13: Aufteilung der Zeiterfassung nach Teammitgliedern

9 Fazit

Die Entwicklung der Rocket App war ein anspruchsvolles, aber äußerst lehrreiches Projekt, das verschiedene Aspekte der modernen Softwareentwicklung praxisnah miteinander verknüpft hat. Von der Planung über die Implementierung bis hin zum Deployment konnten wir nicht nur unser technisches Wissen vertiefen, sondern auch wichtige Erfahrungen in der Teamarbeit, im Projektmanagement und in der Kommunikation sammeln. Besonders hervorzuheben ist die Vielschichtigkeit des Projekts: Die Kombination aus mobiler App, Backend-API, Webplattform und DevOps-Elementen wie Containerisierung und automatisiertem Deployment stellte uns immer wieder vor neue Herausforderungen, die wir gemeinsam lösen konnten. Dabei haben sich Werkzeuge wie GitHub, Docker, Flutter und Go als wertvolle Technologien erwiesen. Neben der technischen Umsetzung stand auch die Nutzererfahrung im Fokus: Durch Features wie Gamification, soziale Interaktion und eine ansprechende UI ist es uns gelungen, eine Anwendung zu schaffen, die funktional und motivierend zugleich ist. Der Einsatz eines flexiblen Architekturdesigns und die Nutzung von Cloud-Infrastrukturen ermöglichen zudem eine langfristige Wartbarkeit und Erweiterbarkeit der App. Abschließend lässt sich sagen, dass die Rocket App nicht nur ein erfolgreiches Ergebnis unseres Projektmoduls darstellt, sondern auch eine solide Basis für zukünftige Weiterentwicklungen bietet. Das Projekt hat uns gezeigt, wie wichtig saubere Planung, strukturierte Umsetzung und kontinuierliches Feedback sind – Fähigkeiten, die wir auch in kommenden Projekten einsetzen werden.

10 Referenzen

References

[1] Android Studio, *Android Studio IDE*, https://developer.android.com/studio

[2] Docker, *Docker: Empowering App Development for Developers*, https://www.docker.com/

[3] Docker Hub zephiron, *Docker Hub: Container Registry*, https://hub.docker.com/u/zephiron

[4] Flutter, *Flutter - Beautiful native apps in record time*, https://flutter.dev/

[5] Flutter Foreground Task, *Flutter Foreground Task Plugin*, https://pub.dev/packages/flutter_foreground_task

[6] Flutter OSM Plugin, *Flutter OSM Plugin for OpenStreetMap*, https://pub.dev/packages/flutter_osm_plugin

[7] Flutter Secure Storage, *Flutter Secure Storage Plugin*, https://pub.dev/packages/flutter_secure_storage

[8] Ginkgo, *Ginkgo: BDD Testing Framework for Go*, https://onsi.github.io/ginkgo/

[9] GitHub, *GitHub: Where the world builds software*, https://github.com/

[10] Gomega, *Gomega: Matcher Library for Go*, https://onsi.github.io/gomega/

[11] Go, *The Go Programming Language*, https://golang.org/

[12] Go Language Server Protocol, *gopls: The Go Language Server*, https://github.com/golang/tools/tree/master/gopls

[13] JWT, *JSON Web Tokens*, https://jwt.io/

[14] Migrate, *Migrate: Database Migration Tool*, https://github.com/golang-migrate/migrate

[15] NGINX, *NGINX Web Server*, https://nginx.org/

[16] OpenTopoData, *OpenTopoData: Open Topographic Data*, https://opentopodata.org/

[17] OpenStreetMap, *OpenStreetMap Project*, https://www.openstreetmap.org/

[18] Pedometer, *Flutter Pedometer Plugin*, https://pub.dev/packages/pedometer

[19] PostgreSQL, *The World's Most Advanced Open Source Relational Database*, https://www.postgresql.org/

[20] Testcontainers, *Testcontainers for Go*, https://golang.testcontainers.org/

[21] Vite, *Vite: Next Generation Frontend Tooling*, https://vitejs.dev/

[22] Vue.js, *Vue.js: The Progressive JavaScript Framework*, https://vuejs.org/

[23] Visual Studio Code, *Visual Studio Code: Code Editing. Redefined*, https://code.visualstudio.com/

[24] Zed, *Zed: A modern code editor*, https://zed.dev/

[25] Everhour, *Everhour: Time Tracking for Teams*, https://everhour.com/

[26] Clockify, *Clockify: Free Time Tracking Software*, https://clockify.me/